MOTOROLA

M68KPASC/D7

M68000 Family
Resident Pascal
User's Manual

MICROSYSTEMS

QUALITY • PEOPLE • PERFORMANCE

M68KPASC/D7

AUGUST 1983

M68000 FAMILY

RESIDENT PASCAL

USER'S MANUAL

Seventh Edition

Sixth Edition December 1982

TABLE OF CONTENTS

TABLE OF CONTENTS (cont'd)

TABLE OF CONTENTS (cont'd)

LIST OF ILLUSTRATIONS

Page

LIST OF TABLES

CHAPTER 1

INTRODUCTION

1.1 SCOPE

This manual explains how a program written in the Motorola Pascal language is readied, compiled, link edited, and run on an EXORmacs Development System, a VMEmodule Monoboard Microcomputer System (VME/10), a VERSAmodule 01 (VM01) or VERSAmodule 02 (VM02) Monoboard Microcomputer, a VMC 68/2 Microcomputer System, or a VMEmodule Monoboard Microcomputer-based System (MVME110). In this manual, a program run under VERSAdos on a VM01, VM02, VMC 68/2, or MVME110 system is referred to as a "VERSAdos01/02" program; a program run under RMS68K on a VM01, VM02, VMC 68/2, or MVME110 system is referred to as an "RMS68K01/02" program.

1.2 GENERAL

Pascal, first developed as a teaching tool, has gained wide acceptance as an applications and systems programming language. Its structured nature and ease of maintenance have made it a favorable language in saving time and effort for users in program development/support. In recognition of this, Pascal has been developed by Motorola as one of the high-level programming languages to be used on the M68000/M68010 systems.

1.3 OPERATING ENVIRONMENT

- VERSAdos Operating System linked with the appropriate Pascal runtime library. (See Chapter 5.)

1.4 MINIMUM HARDWARE REQUIREMENTS

1.4.1 EXORmacs Development System

- EXORmacs Chassis
- EXORterm 155 Display Console
- EXORdisk III Disk Drive Unit
- Model 703 Printer, or equivalent
- VM01 Monoboard Microcomputer (optional)
- VM02 Monoboard Microcomputer (optional)
- MVME110 Monoboard Microcomputer (optional)
- 384K Bytes of RAM (includes ample user RAM)

1.4.2 VMC 68/2 Microcomputer System

- VMC 68/2 System (which includes an MLD-16 8" LARK disk drive unit, and 384K bytes of RAM)

- EXORterm 155 Display Console, or user-supplied dumb ASCII RS-232C terminal

- Model 703 Printer, or equivalent (optional)

1.4.3 VME/10 Microcomputer System

- VME/10 System (which includes Control Unit Chassis, Display Unit, Keyboard, and 384K bytes of RAM)
- Model 703 Printer, or equivalent (optional)

1.5 RELATED PUBLICATIONS

- EXORmacs Development System Operations Manual (M68KMACS)
- M68000 Family Real-Time Multitasking Software User's Manual (M68KRMS68K)
- M68000 Family System Generation Facility User's Manual (M68KSYSGEN)
- VERSAdos Data Management Services and Program Loader User's Manual (RMS68KIO)
- VERSAdos Overview (M68KVOVER)
- VERSAdos System Facilities Reference Manual (M68KVSF)
- M68000 Family Linkage Editor User's Manual (M68KLINK)
- M68000 Family CRT Text Editor User's Manual (M68KEDIT)
- MC68000 16-Bit Microprocessor User's Manual (MC68000UM)
- M68000 Family Resident Structured Assembler Reference Manual (M68KMASM)
- M68KVM01-1, -2 Monoboard Microcomputer User's Guide (M68KVM01)
- VERSAbug Debugging Package User's Manual (M68KVBUG)
- Pascal Programming Structures for Motorola Microprocessors
- MC68010 16-Bit Virtual Memory Microprocessor product specification handbook (ADI-942)
- VMC 68/2 - Series Microcomputer System Manual (MVMCSM)
- VME/10 Microcomputer System Overview Manual (M68KVSOM)

1.6 GENERAL STRUCTURE OF PASCAL COMPILER

As shown in Figure 1-1, the Pascal compiler may consist of two or three phases. Phase 1 processes a source program and produces a compilation listing and error messages, as well as an intermediate code file. Optionally, Phase 1.5 may be invoked in order to "optimize" the intermediate code -- i.e., to reduce the size of the code file. Either the Phase 1 intermediate code or the Phase 1.5 optimized intermediate code is then input to Phase 2 in order to create a relocatable object file, as well as a listing. The object file may be combined with needed routines from the runtime library by the M68000 Family Linkage Editor (also referred to as the linker) to create a load module which is ready to run. Alternatively, the linker's output may be a relocatable object module which can be linked with others, or a transportable S-record format module. (Linking is discussed in Chapter 5.)

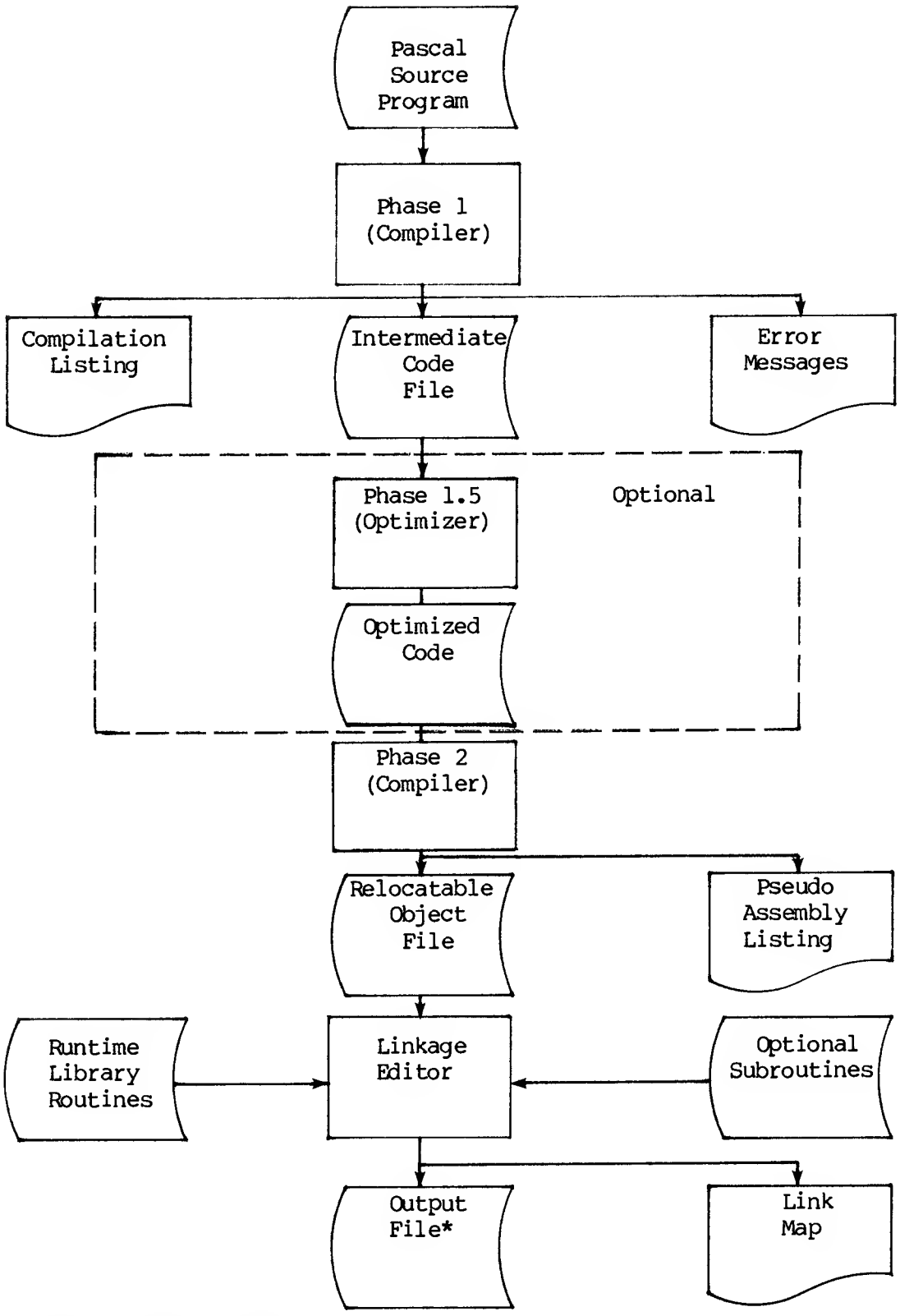
If the module is to be run on VERSAmodule 01 under VERSAbug, it must be an S-record file linked with the VERSAmodule 01 runtime library. Paragraph 5.8 shows one method of accomplishing this, and paragraph 7.2 shows a method of downloading and executing a program on the VERSAmodule 01 under VERSAbug.

1.7 NOTATION AND RULES

Commands and other input/output (I/O) are presented in this manual in a modified Backus-Naur Form (BNF). Certain symbols in the syntax may be used, where noted, in the real I/O; however, others are meta-symbols and their meanings are as follows:

< >	Angular brackets enclose a symbol, known as a syntactic variable, that is replaced in a command line by one of a class of symbols it represents.
|	This symbol indicates that a choice is to be made. One of several symbols, separated by this symbol, should be selected.
[ ]	Square brackets enclose a symbol that is optional. The enclosed symbol may occur zero or one time.
[ ]...	Square brackets followed by periods enclose a symbol that is optional/repetitive. The symbol may appear zero or more times.
::=	Two colons followed by an equal sign means "may be composed of".

Commands entered at the keyboard are to be terminated by pressing the RETURN key.



*Relocatable Object Module, Load Module, or S-Record Module

FIGURE 1-1. Pascal Program Processing

CHAPTER 2

PREPARING A PROGRAM

2.1 THE OPTION COMMENT

The Pascal source program may include options (Table 2-1) that affect the compiler's source and object output, options that control runtime checks, and miscellaneous options. The options are named in an option comment, which is enclosed within braces (i.e., { }) or within the symbol pairs (* and *). A dollar sign immediately follows the left brace or left symbol pair to identify the comment as an option comment. The format of the option comment is:

{$Xs,...,Xs}

or

(*$Xs,...,Xs*)

where X is a capital letter corresponding to one of the options shown in Table 2-1. The s is either a plus (+) or a minus (-) sign. A plus assigns a TRUE value enabling the option, and a minus assigns a FALSE value disabling the option. For the page eject option, E, the sign is omitted. The sign is also omitted for options in which a number is needed (X=n). Table 2-1 shows the default value assigned to each option at the beginning of the program.

One or more options, separated by commas with no intervening spaces, may be specified in the comment. The option comment may appear at any point in a program at which a comment is normally allowed (except the Q option, which must appear prior to any keyword except 'program' or 'subprogram'). Other text may be added to an option comment, provided it is separated from the options by at least one blank space. These options are similar to those allowed in the command line when invoking the compiler (Section 4). Except for option Q, command line options do not override options specified in source comments.

2.2 FILE NAME FORMAT

In this description, some syntactic variables are defined by the term "file name". A VERSAdos disk file name consists of six fields:

<volume name>:<user number>.<catalog>.<file name>.<extension>(<protect key>)

If any of these fields are omitted, the system will fill them in with default values, as follows:

a. If <volume name> is omitted, the volume specified at system logon, or in the last session control USE command, or specified in the first command parameter (overrides defaults) will be used.

b. If <user number> is not supplied, the user number supplied at logon, or in the last USE command, or specified in the first command parameter (overrides defaults) is the default.

c. If a default catalog has been supplied at logon, or with the USE command, or specified in the first command parameter (overrides defaults), and <catalog> is not specified, then the default catalog will be used. Any <catalog> specified will override the default catalog. If a default catalog has been specified and a null catalog is required, entering a & (ampersand) as the <catalog> will produce a null catalog. If a default catalog has not been specified and <catalog> is omitted, then a null catalog will be used.

d. If <user number> and/or <catalog> is being specified, <file name> may be omitted and will default to the file name specified by the first command parameter.

e. If the <extension> field is not supplied, a default will be supplied. See command line descriptions in Chapter 4 for default extensions.

f. If <protect key> -- 2- to 4-character (AA-PP) access protection code -- is not specified, it defaults to PPPP (any user may read or write to the files). If only two characters are specified, they are assumed to be the read code and the write code defaults to public write (PP).

The following file names are equivalent if the last USE command specified VOL1 and the user logged on as user 3:

```
VOL1:3..TESTPROG.SA
3..TESTPROG.SA
TESTPROG.SA
TESTPROG        (if default extension is .SA)
```

TABLE 2-1. Source Program Options

OPTION	DEFAULT VALUE	MEANING
A=n	A=4	Specify the number of bytes -- where n is 1, 2, or 4 -- used for integer arithmetic. If n is 1, one-byte arithmetic is performed; if n is 2, two-byte arithmetic is performed; and if n is 4, four-byte arithmetic is performed.
C-	C+	Generate an intermediate code file during Phase 1. If C- is specified, an intermediate code file is not generated. (Eliminating this file reduces the time necessary to generate the listing and any errors.)
D+	D-	This combines the K and R options to (1) generate code to perform runtime checks which verify that array indices and subrange type variables are in range, and (2) include executable unit numbers in the executable object code. The numbers relate to statements and are found on the source listing. If an error condition occurs at runtime, the current executable unit number is contained in register D1.

TABLE 2-1. Source Program Options (cont'd)

OPTION	DEFAULT VALUE	MEANING
E		Page eject. Whenever this option is encountered in the source program, the Phase 1 listing will advance to the top of the next page. (This option has no default value, and no plus or minus sign.)
F=<fn>		Include the file specified by <fn> in the source. Immediately after the comment which contains this option, Phase 1 will start obtaining its source input from the file indicated by <fn> (which must conform to the rules for specifying a file name for the operating system). When the end of the "include file" is encountered, Phase 1 will return to getting its source from the original source file at the point it left off. There are three restrictions on the use of this option: a) When the end of the comment containing the F option is encountered, the remainder of that source line must not contain any more text. If more source text is found, it will be ignored. b) Only one F option may be specified per comment. c) Include files may not be nested - that is, a source file that is being read as the result of an F option may not itself contain an F option. Thus, the only file that may contain F options is the original source file.
G+	G-	Keep files output by the compiler or optimizer which contain errors. In the default condition, G-, such erroneous files are deleted.
H=n	H=4096	n is the size of the program heap in bytes. n has the format: [<base>#]<integer>. If "<base>#" is omitted, <integer> defaults to a decimal base; otherwise, the symbol # separates the base and integer; <base> is written in decimal, and <integer> is written in the base. <base> may range between 2 and 16, inclusive.
I-	I+	Pass any external files specified on the command line to the program at start-up. If I- is specified, the command line will not be scanned for external files. In this case, only the Z option (paragraph 7.1.2) is recognized.

TABLE 2-1. Source Program Options (cont'd)

OPTION	DEFAULT VALUE	MEANING
K+	K-	Include executable unit numbers in the executable object code. The executable unit numbers relate to statements and are found on the source listing. If an error condition occurs at runtime, the current executable unit number is contained in register D1. Note that there is a code overhead of 2 bytes or 6 bytes each time the executable unit counter is updated.
L-	L+	Generate a source listing on Phase 1 listing file/ device (printer or CRT).
O+	O-	Enter source statements as comments in the Phase 2 input.
P+	P-	Include executable unit numbers in the executable object code, but only at function/procedure entry and exit points.
Q+	Q-	Use fast floating point. In the default condition, Q-, fast floating point is disabled. When using separate compilation, the same state of this option (Q+ or Q-) must be used with each compilation. If specified, this option must appear prior to any keyword except program or subprogram.
R+	R-	Generate code to perform runtime checks which verify that array indices and subrange type variables are in range.
S=n	**	The value specified by n (whose format is described under the H option above) will be the default stack/heap size in bytes used by the program. If specified, n must be at least 768. ** If the S option is zero or is not specified, the default stack/heap size in bytes is the following summation: (Heap size, specified by the H option) + (Size of the global variable area) + (Runtime maintenance area and vector table size: 768 bytes) + 1.5 x (P1 + P2 +...+ P7) where Pi is the maximum size of all the local variable areas in procedures declared at level i.
W+	W-	Generate a warning during Phase 1 processing if non-standard Pascal features are used. Standard Pascal comprises only the language features proposed by Jensen and Wirth.

CHAPTER 3

FUNCTIONAL DESCRIPTION

3.1 GENERAL

A Pascal source program prepared by the user must be processed by the M68000 Family Pascal Compiler to produce a relocatable object file, from which an executable load module can be created.

The M68000 Family Pascal Compiler, referred to as the "compiler", consists of three programs. The first of these, Phase 1 of the compiler, is invoked using the PASCAL command. When the first phase completes, the user activates Phase 2 via the PASCAL2 command or, optionally, may invoke Phase 1.5 with the POPTIM command to optimize the intermediate code produced by Phase 1, and then run Phase 2. Optimized code allows faster execution of compiled, linked programs. During each of the three phases, progress counters displayed on the terminal are updated every 100 source lines and at the end of the code in the input file. The counter update overwrites the old values with new values. Counter updating can be disabled for any or all of the three phases at the user's option (option E-). The output produced by Phase 2 must then be processed by the linkage editor, described in Chapter 5 of this manual.

3.2 COMPILER, PHASE 1

Phase 1 processes a Pascal source program, checking the syntax of each statement it encounters. If any errors are detected, they are brought to the attention of the user. These errors should be eliminated and Phase 1 should again be invoked to compile the modified program. When no errors are reported, Phase 1 processing is complete.

Phase 1 of the compiler produces two types of output. First, it generates an intermediate file which is used to produce the relocatable object file from Phase 2. If errors were detected during Phase 1 processing, the intermediate file is of no value and is automatically deleted unless the G option is on.

Second, it produces an optional listing of the source program containing error codes along with other useful information. When an error is detected, a line is added to the program listing containing the phrase "***Error---" followed by the line number of a previous error or by 0 if this is the first error. Also on this line appears an error code positioned beneath the symbol that was being processed when the error was discovered.

Each line of the source listing file contains the following fields:

LINE — Source program line number. Up to five digits may appear in this field.

LOC — LOC stands for location. If enclosed in parentheses, this field contains the offset in the data section of the first variable declared in this statement; otherwise, this field contains an executable unit number, roughly corresponding to a statement number. If an error condition occurs while the program is running and a debug option (D or K) was selected, the executable unit number of the statement being processed will be contained in register D1 to indicate the point of failure.

LEV — LEV stands for level. Level numbers indicate the static structure of a program. The main program is at level 0. A level 1 procedure is contained in the main program and in no other procedure. A level n procedure is contained by procedures at level 0 through n-1. Level numbers are useful when determining the scope of variables or procedures.

B — B is an abbreviation for block beginner. A block beginner is one of the following symbols: BEGIN, REPEAT, or CASE. When one of these keywords is encountered, the B level is incremented. If multiple keywords that increase the B level occur on one line, the level corresponding to the first beginner is printed.

E — E stands for block terminator. A block terminator is either of the symbols: END or UNTIL. An END will match either an earlier BEGIN or a previous CASE symbol. An UNTIL is always associated with an earlier REPEAT. The E level is decremented when a block terminator is processed. If multiple block terminators are encountered in a line, the level of the last block terminator is printed.

Block levels are described by increasing letters of the alphabet. If a block beginner does not appear in a line, its B field contains a dash (-); if no block terminator is found on a line, its E field is also a dash. The B and E fields enable the user to quickly determine the block structure of a program. A common error is to fail to provide a matching block terminator for each block beginner. Often an examination of these fields will pinpoint the location of the error.

- — The remaining field contains a copy of the source statement, truncated to the current line length.

At the end of the listing, a summary of the compilation is provided. A count of syntax errors, warnings, lines of source, procedures, and P-codes (intermediate code instructions) is given. If any errors or warnings occurred, the line number of the last error is listed.

An example of a source program listing containing two errors is shown in Figure 3-1. This figure shows how lines containing errors are chained together and also illustrates the other fields described above.

```
Line      Loc Lev BE Motorola Pascal          FIB8     .SA

   1(      -8) 0)— program Fibonacci(output);
   2(      -8) 0)—
   3(     -20) 0)—      var a,b,i : integer;
   4(       0) 1)— procedure fib(var x,y: integer);
   5(      -4) 1)—      var temp : integer;
   6        1  1)A-           begin (* FIB *)
   7        2  1)—                tmp := y; (* compute the next Fibonacci *)
***Error——     0***                  ^104
   8        3  1)—                y := y + x (* number (f(n), f(n-1) —> (f(n-1,f(n))*);
   9        4  1)—                x := temp
  10           1)-A          end; (* FIB *)
  11        5  0)A-          begin
  12        6  0)—                a := 0; (* initialize a and b*)
  13        7  0)—                b := 0;
  14        8  0)—                   for i := 2 to 10 do DO
***Error——     7***                                    ^6
  15           0)B-                  begin
  16        9  0)—                      fib(a,b);
  17       10  0)—                      writeln(output,i:3,b:5)
  18           0)-B                  end
  19           0)-A          end.   (* Fibonacci *)

      **** 2 Error(s) and No Warning(s) detected

      **** Last error line was 14

      **** 19 Lines 1 Procedures

      **** 14 Pcode instructions
```

FIGURE 3-1. Pascal Listing with Errors

3.3 OPTIMIZER, PHASE 1.5

Phase 1.5 is an optional phase which may be chosen by the user. This phase provides machine-independent optimization of the pseudo-code produced by the compiler by reducing the number of pseudo-codes and providing more information to the machine-dependent code generator about the program in general and variable usage in particular.

Upon successful completion of Phase 1.5, the optimized intermediate code has been written to an output file which, in turn, becomes the input file for Phase 2 of the compiler. If errors are detected during the optimizer pass, however, the output file is automatically deleted, unless the G option is on.

3.4 COMPILER, PHASE 2

Phase 2 of the compiler processes the intermediate code produced by Phase 1 or the optimized intermediate code produced by Phase 1.5, and generates an object module that can be link edited to create a load module. It then generates, in the form of a relocatable object module, the machine code equivalent of the corresponding group of intermediate instructions. One object module is generated for the entire input file.

Phase 2 of the compiler optionally produces a pseudo assembly listing (option L+). The listing is normally not needed and may be suppressed by the user.

During Phase 2 compilation, a count of bytes of code generated is output to the console. These "progress counters" are updated periodically, unless option E is on. When Phase 2 processing completes, the total count is shown along with a message indicating whether or not errors in generated code were detected. If errors were detected, the object output file is automatically deleted (unless the G option is on).

3.4.1 Object File Description

The compiler produces a relocatable object file that is compatible with the M68000 Family Linkage Editor. The object module contains information which, when extracted by the linker, makes possible the combination of separate programs and the automatic inclusion of necessary system routines. The location of every level 1 procedure is recorded in the object file in an external definition record. A list of all modules referenced by the program, either explicitly requested by the user or determined by Phase 2 to be needed, is included in an external reference record. An indication of the memory occupied by the program is provided, along with a request for space to be used by the Pascal program for data storage in a stack/heap.

The code itself is also stored in the object module. Phase 2 creates code that is position independent, as well as relocatable. The linking process will preserve the position-independence so that Pascal programs may theoretically be loaded into any memory address space. A special feature of this code is that it includes a pseudo long relative branch facility that enables any instruction to be reached with six bytes of code. Routines obtained from the runtime library may always be reached with a four-byte instruction.

3.4.2 Pseudo Assembly Listing Description

If a Pascal program does not perform as expected, debugging may be necessary. The most convenient way to perform this activity is by including facilities in the program to inform the user of its progress, reporting the values of critical variables at appropriate times. Occasionally it might be desirable to conduct debugging of individual machine instructions rather than source statements. The pseudo assembly listing greatly facilitates this activity.

This listing contains the following information:

a. Pascal source statements are present if the O option was selected when Phase 1 processing was requested. To the right of the source statement appears a statement number that matches the statement number appearing at the beginning of each line of the Phase 1 listing. This makes it easy to find a specific source statement in the pseudo assembly listing.

b. Between source statements appears a representation of the code that was stored in the object file. This appears in a similar format to that which would be produced by an assembler. Machine code for instructions which cannot be shown in final form (instructions containing forward references and instructions requiring linkage for completion) is indicated by asterisks (**).

c. An assembly language instruction equivalent to the machine code representation appears on the right side of the pseudo assembly listing. This code may serve as a basis for users desiring to modify code generated by Phase 2, but will not, in general, assemble correctly.

d. In certain situations, addresses have not been determined at the time the listing is generated. In the Phase 2 listing, unknown addresses jumped to or branched to are indicated by asterisks. Instruction addresses which are uncertain at this time are shown as ranges in which they will fall -- e.g., 00000054-005C. This uncertainty results from forward references to labels and Phase 2's attempt to reach the label using a short branch. Phase 2 does not know whether a short branch will be adequate until sometime after the pseudo assembly listing has been output.

CHAPTER 4

EXECUTING THE PASCAL COMPILER

4.1 INTRODUCTION

The following paragraphs describe the commands used to run the three parts of the Pascal compiler -- Phases 1, 1.5, and 2, in turn. These are three separate disk-resident programs named PASCAL, POPTIM, and PASCAL2, respectively. Phase 1.5, the optimizer (POPTIM), is optional and is invoked where the fastest, most efficient object code is desired in the completed compiled, linked program. Phase 1 is run first to compile the source code into "pseudo" code (P-code); the output of Phase 1 may be input directly to Phase 2, which accepts the output of either Phase 1 or of Phase 1.5. The relocatable object file produced by Phase 2 is then linked to produce an executable load module (Chapter 5).

4.2 RUNNING PHASE 1

Phase 1 of the Pascal Compiler is loaded and run in response to the following VERSAdos command line:

PASCAL <fn1>[/<fn1>]...[,[<fn2>][,<fn3>]][;<options>]

where the syntactic variables are defined as follows:

fn1 — Input file(s). These are names of disk files (one or more), which contain the Pascal source program. As many input files as desired may be specified on the command line. Extensions, if not given, default to .SA.

fn2 — Output file. This is the name of the disk file which will contain the intermediate code created by Phase 1. If not specified, it assumes the name of the first input file processed, but with an extension of .PC.

fn3 — Listing file. This is the name of the disk file which will be used to contain the Phase 1 listing. If not specified, it assumes the name of the first input file processed, but with a default extension of .PL. If # or #PR is specified instead of <fn3>, the listing will be directed to the user's console or line printer, respectively. If #NULL is specified, no listing at all is generated.

options — The options which may be specified on the command line are similar to those which may be specified in an option comment in the source program (see Table 2-1). Except for option Q, command line options do not override those in a program's option comment. Note that the minus sign (-) precedes the option letter when used on the command line.

OPTION	DEFAULT	
C	C	Generate an intermediate code file. -C suppresses its generation.

OPTION	DEFAULT	
D	-D	Generate runtime range checking code; include executable unit numbers in object code. Both functions are suppressed if -D.
E	-E	Do not update progress counters. The default value, -E, indicates the counters are to be updated.
G	-G	Retain the intermediate code file in the event an error is detected. The default value, -G, deletes this file if an error is detected.
I	I	Pass external files from the command line. -I suppresses this function.
K	-K	Include executable unit numbers in object code. -K suppresses the inclusion of the numbers in the object code.
L	L	Generate a Phase 1 listing. -L suppresses generation of the listing; only those lines containing compile time errors will be displayed.
O	-O	Include source statements in Phase 2 input. -O suppresses this function.
P	-P	Include executable unit numbers in object code at function/procedure entry and exit points. -P suppresses this function.
Q	-Q	Use fast floating point. If -Q, fast floating point is not used. When using separate compilation, the same state of this option (Q or -Q) must be used with each compilation. If the program being compiled contains an option comment (Q+ or Q-), a 514 error message will be issued (floating point type already specified).
R	-R	Generate runtime range checking code. -R suppresses generation of the runtime range checking code.
W	-W	Warn if non-standard Pascal features are used. Default (-W) is no warnings.
Z=n	Z=40	Set stack/heap (symbol table) size used by compiler to nK. Value of n must be at least 40 (the default value, 40K bytes). The default value will be adequate for compiling most programs. However, some larger programs may cause Phase 1 to abort with error 1008, 1010, or 1011. In such cases, Phase 1 should be executed with a larger Z option.

4.2.1 Phase 1 Acknowledgment

When the command line to Phase 1 has been correctly entered, the following acknowledgment is displayed on the screen:

```
Motorola Pascal Compiler Phase 1 Version x.xx
Copyrighted 1982 by Motorola, Inc.

Source Lines    Intermediate Code    Errors    Warnings
   nnnnn            nnnnnn             nnn       nnn
```

The counters are updated every 100 source lines processed during the compilation (unless disabled by the E option). Upon completion, final counter values are always displayed.

4.2.2 Command Line Examples

The following command lines are equivalent:

```
PASCAL    TESTPROG
PASCAL    TESTPROG,TESTPROG,TESTPROG
PASCAL    TESTPROG.SA,TESTPROG.PC,TESTPROG.PL
PASCAL    TESTPROG,.PC,.PL
```

All of the above commands direct Phase 1 to process a source program contained in TESTPROG.SA and produce intermediate code in TESTPROG.PC and a listing in TESTPROG.PL. If errors are detected, however, the file TESTPROG.PC will be deleted (assuming the default option -G).

A common form of the command is:

```
PASCAL    TESTPROG,,#;-L
```

This command compiles TESTPROG.SA, creates intermediate code in TESTPROG.PC, and displays only lines containing compile time errors and associated error messages on the console screen.

When Phase 1 is executed from a CHAIN or BATCH file, upon termination it will load the diagnostic register (RD) with a value reflecting the success of the compilation. A value of $0000 indicates no warning or error conditions were detected; a value of $1nnn indicates that $nnn warning conditions were detected; a value of $Cnnn indicates that $nnn errors were detected. Note that a value of this last form will normally cause the CHAIN or BATCH file to abort.

4.3 RUNNING PHASE 1.5

Phase 1.5, the optimizer, is loaded and run in response to the following VERSAdos command line:

```
POPTIM <fn1>[,<fn2>][;<options>]
```

where the syntactical variables are defined as follows:

fn1 — Input file. This is the name of the disk file containing the intermediate code generated by Phase 1. The extension, if not given, defaults to .PC.

fn2 — Output file. This is the name of the disk file which will contain the optimized intermediate code. If not specified, it assumes the name of the input file, but with an extension of .PO.

options — May be one or more of the following:

OPTION	DEFAULT	
E	-E	Disable progress counter updating during optimization. The default value, -E, enables the counter updating.
G	-G	Retain the optimized intermediate code file in the event an error is detected. The default value, -G, deletes the optimized intermediate code file if an error is detected.
O=n	O=1	Perform levels of optimization up to n. If specified, n must be in the range of 1 to 3. The default level is 1. (Levels 2 and 3 are not yet implemented.)
Z=n	Z=32	Allocate a stack and heap segment of size at least nK (1K = 1024 bytes) for the optimizer during the processing of the input file. The output file is not affected by this option. The default size will be adequate for most programs. However, some larger programs may cause Phase 1.5 to abort with error 1008, 1010, or 1011. In such cases, Phase 1.5 should be executed with a larger stack/heap size.

4.3.1 Phase 1.5 Acknowledgment

When the command line to Phase 1.5 has been entered correctly, the following acknowledgment is displayed:

```
Motorola Pascal Optimizer Version x.xx
Copyrighted 1982 by Motorola, Inc.

Source Lines    Intermediate Code    Optimized Code
   nnnnn            nnnnnn               nnnnnn
```

The counters are updated approximately every 100 source lines processed during optimization (unless disabled by the E option). Upon completion, final counter values are always displayed.

4.3.2 Command Line Examples

```
POPTIM    TESTPROG
POPTIM    TESTPROG,TESTPROG
POPTIM    TESTPROG.PC,TESTPROG.PO
POPTIM    TESTPROG;O=1,-G
```

All of the above commands cause the intermediate code in a file named TESTPROG.PC to be optimized up to level 1 of the optimizer, then for the optimized intermediate code to be placed in a file named TESTPROG.PO. If errors are detected, however, the file TESTPROG.PO will be deleted (default option -G).

When Phase 1.5 is executed from a CHAIN or BATCH file, upon termination it will load the diagnostic register (RD) with a value reflecting the success of the optimization. A value of $0000 indicates no warning or error conditions were detected; a value of $1000 indicates that warning conditions were detected; and a value of $C000 indicates that error conditions were detected. Note that a value of $C000 will normally cause the CHAIN or BATCH file to abort.

4.4 RUNNING PHASE 2

Phase 2 of the Pascal Compiler is invoked with the following VERSAdos command:

```
PASCAL2   <fn1>[,[<fn2>][,<fn3>]][;<options>]
```

where the syntactic variables are defined as follows:

fn1 — Input file. This is the name of the disk file containing the intermediate code generated by Phase 1 or the optimizer. The extension, if not given, defaults to .PC. If the file has been optimized, the optimized extension (default .PO) must be specified.

fn2 — Output file. This is the name of the disk file which will contain the relocatable object module. If not specified, it assumes the name of the input file, but with an extension of .RO.

fn3 — Listing file. When the L option is specified, this is the name of the disk file which will contain the listing produced by Phase 2. If name is not given, it assumes the name of the input file, but with an extension of .LS.

If # or #PR is specified instead of <fn3>, the listing will be directed to the user's console or line printer, respectively.

options — May be one or more of the following:

OPTION	DEFAULT	
E	-E	Do not update progress counters. The default value, -E, indicates the counters are to be updated.
G	-G	Retain the relocatable object output file in the event an error is detected. The default value, -G, deletes this file if an error is detected.

OPTION	DEFAULT	
L	-L	This option enables generation of the listing file specified by <fn3>. -L, the default value, suppresses generation of the listing file, thus making it possible to generate the output file more quickly.
J	J	This option causes a JSR to an F-line trap simulator to be generated before each floating point instruction generated when the standard version of floating point is being used (-Q). This is the default condition. Entering -J as a command line option suppresses generation of the JSR's to the F-line trap simulator. If -J is used, the user must supply his own floating point initialization routine, F-line trap handler, and memory access routine at linkage edit time. These are described in Chapter 10. When using separate compilation, the same state of this option, J or -J, must be used with each compilation.
Z=n	Z=48	Set stack/heap size used by compiler to nK. Value of n must be at least 48. The default value will be adequate for most programs. However, some larger programs may cause Phase 2 to abort with an error 1008, 1010, or 1011. In such cases, Phase 2 should be executed with a larger Z option.

4.4.1 Phase 2 Acknowledgment

When the command line to Phase 2 has been entered correctly, the following acknowledgment is displayed:

```
M68000 Pascal Compiler Phase 2 Version x.xx
Copyrighted 1982 by Motorola, Inc.

Source Lines    Intermediate Code    Bytes Generated
   nnnnn            nnnnnn               nnnnnn
```

The counters are updated every 100 source lines processed during code generation (unless disabled by the E option). Upon completion, the final counter values will always be displayed.

4.4.2 Command Line Examples

Since the listing file output by Phase 2 is normally not needed, it is suppressed by default. For example:

```
PASCAL2  TESTPROG
```

This command processes the intermediate code in TESTPROG.PC, creates an object module in the file TESTPROG.RO, and produces no listing.

Another example shows how the input and output may be on different volumes:

```
PASCAL2   TESTPROG,VOL1:
```

The above example processes TESTPROG.PC and creates a relocatable object module on VOL1 called TESTPROG.RO. No listing is produced.

To run Phase 2 on the optimized code output by Phase 1.5, the file name's extension must be specified. For example:

```
PASCAL2   TESTPROG.PO
```

processes the optimized code in file TESTPROG.PO and creates a relocatable object module in TESTPROG.RO. No listing is produced.

A final example shows how a listing is produced:

```
PASCAL2   TESTPROG;L
```

The above example processes TESTPROG.PC, creates a relocatable object module in TESTPROG.RO, and generates a listing in TESTPROG.LS.

In all the above examples, if errors are detected the .RO file is deleted (assuming option default -G).

When Phase 2 is executed from a CHAIN or BATCH file, upon termination it will load the diagnostic register (RD) with a value reflecting the success of the code generation. A value of $0000 indicates that no warning or error conditions were detected; a value of $1000 indicates that warning conditions were detected; and a value of $C000 indicates that error conditions were detected. Note that a value of $C000 will normally cause the CHAIN or BATCH file to abort.

Phase 2 uses a scratch file during processing. This file is written on the user default volume and is normally deleted at the completion of Phase 2 execution.

CHAPTER 5

RUNNING THE LINKAGE EDITOR

5.1 GENERAL

Relocatable object modules, generated by Phase 2 of the compiler, are processed by the M68000 Family Linkage Editor to produce an absolute load module, an S-record module, or another relocatable object module. A Pascal program requires the linker because:

a. Every Pascal program refers to routines which reside in the runtime library.

b. If a program is to be combined with one or more subprograms that were compiled separately, the linkage between modules must be constructed.

c. If a Pascal program calls a procedure or function written in assembly language, the load module must include object modules produced by the M68000 Family Assembler.

In all these cases, the linker is required to assign memory space (MMU segments for EXORmacs or VME/10, physical memory for VM01, VM02, and MVME110) to each required object module, enable intermodule communication, and create a load module that is ready to run.

5.2 RUNTIME ROUTINES

The Pascal runtime library (PASCALIB) provides certain standard functions that may be optionally used to perform general services. A group of functions and procedures is also provided, which interfaces the Pascal program with the operating system to provide for input or output. A routine is provided to establish the environment required by a Pascal program. Some frequently requested code sequences that perform such activities as manipulating strings or vectors are implemented as runtime routines to reduce program code size.

Whenever a reference is made to one of the runtime routines, an external reference record is produced by the compiler as part of the object module. The linker will include only referenced runtime routines in the load module.

Note that the Pascal compiler assumes that all runtime routines will be stored contiguously in the load module. The user who requests other than default linkage editor processing (paragraph 5.7) must ensure that this assumption is not invalidated.

5.3 SEPARATE COMPILATION

Pascal supports separate compilations so that the user may group one or more procedures or functions into a subprogram. The linker can combine as many subprograms as desired. The locations of all level 1 procedures are made known to the linker by external symbol definition records within the object module. The linker can thus resolve references between the program and subprogram or between two subprograms.

As previously stated, modules using standard floating point may not be linked with modules using fast floating point. If an attempt is made to do so, the link will report that the symbol .PFLOATP is multiply defined. The link will not abort; however, the resulting output module will be invalid.

5.4 ASSEMBLY LANGUAGE PROCEDURES

Pascal permits the user to refer to procedures or functions written in assembly language. If such routines are required, they should be written as shown in Chapter 6. The linker will enable any Pascal program or subprogram to utilize assembly language routines.

5.5 PASCAL PROGRAM MEMORY ORGANIZATION

Pascal organizes programs so that code modules occupy section 9 and the stack and heap use space in section 15. The Pascal runtime routines are allocated space in section 8. The runtime system for the VERSAmodule 01 under VERSAbug uses section 0 to hold data for the various device drivers.

According to the linker's default processing, memory will be allocated in two segments. Segment SEG1, the program segment, will contain the runtime routines, the Pascal code section, and assembly language routines. Segment SEG2, the data segment, will contain the Runtime Maintenance Area (RMA), Pascal stack/heap, and the Pascal exception vectors. For the VERSAmodule 01 under VERSAbug, runtime device data will also be in SEG2 (specified with the linker's SEG command; default link processing for VM01 will result in a non-executable load module; refer to paragraph 5.8). Figure 5-1 shows the memory allocation for a typical Pascal program. Note also that if a Pascal program references an origined variable, it will not be automatically included by the linker.

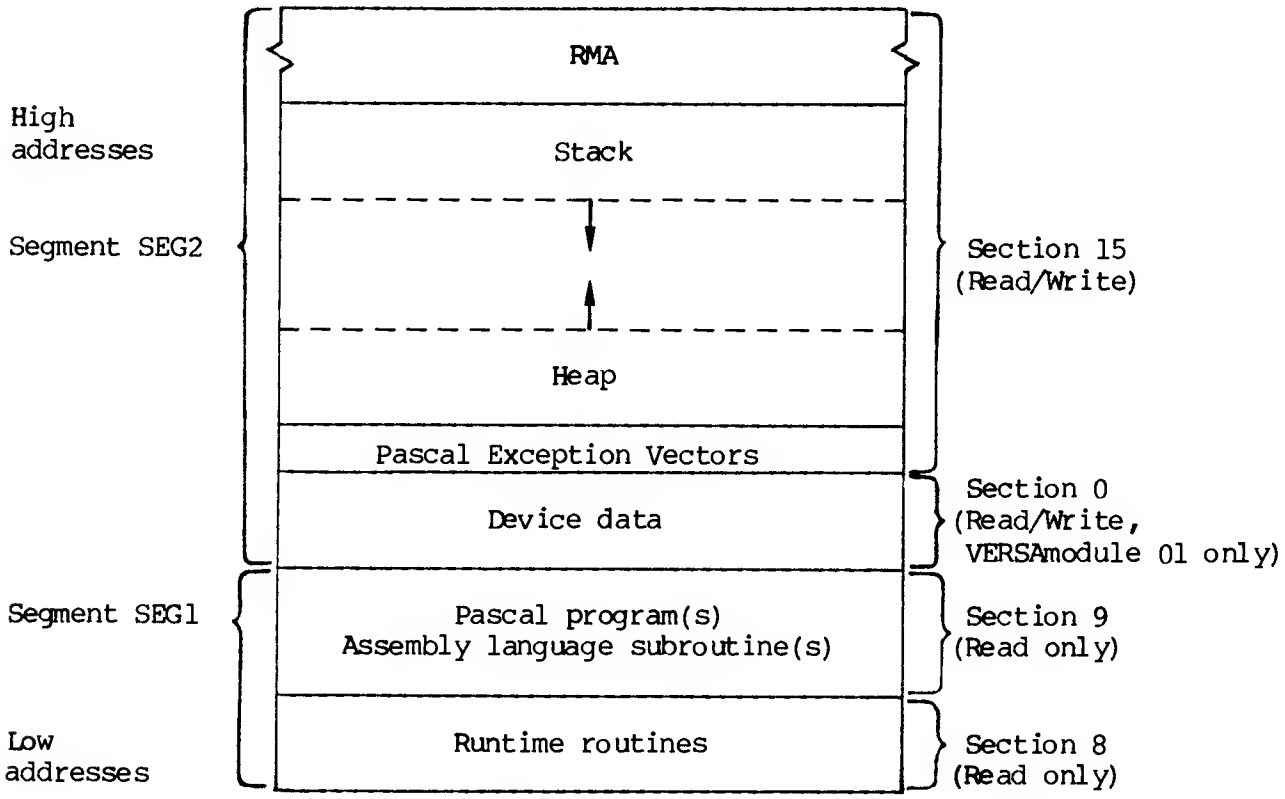


FIGURE 5-1. Pascal Memory Allocation

5.6 STACK AND HEAP USAGE

While a Pascal program is running, two types of memory allocation are used: a stack and a heap.

Variables that are global or local and first appear in a VAR statement are allocated space on the stack. Global variables -- i.e., those declared in the main program -- occupy space for the duration of the program run. Local variables are allocated stack space when the procedure or function in which they are declared is entered, and relinquish their space on the stack when their containing routine is exited.

Variables first appearing in a NEW statement are not stored on the stack, but occupy space on the heap. An appropriate amount of space is allocated on the heap whenever a NEW statement is processed during program execution. This space is not relinquished until a DISPOSE statement is executed.

The stack is built in the highest address of the data segment and grows toward lower addresses; the heap grows from lowest addresses toward higher addresses. The stack and heap may share the data segment space in any ratio, but their total space requirements at any given time must not exceed the total segment size or the program will generate a stack/heap overflow error code and abort or other error codes -- such as 1008, 1010, or 1011 -- and the program will abort (see Appendix D).

The stack/heap size is determined by the options H and S in the option comment of a source program (paragraph 2.1), or by the option Z when used in a runtime command line (paragraph 7.1) or by default calculation if neither H, S, or Z options are specified.

5.7 INVOKING THE LINKER TO CREATE A VERSAdos PROGRAM FOR EXORmacs OR VME/10

A VERSAdos (for EXORmacs or VME/10) load module resulting from the processing of a Pascal program will have a default set of attributes compatible with most users' needs. Default linker processing is requested by a simple LINK command, whose format is as follows:

```
LINK  [[<fn1>[/<fn1>]...][,[<fn2>][,<fn3>|#|#PR]]][;<options>]
```

fn1 — The file name of either a Pascal object module generated during Phase 2 or an assembly language object module. One or more file names may be specified.

If no input files are specified, the linker will prompt for entry of user commands, at which time file names can be specified with the INPUT command.

fn2 — The file name for the load module or S-record module to be generated. This parameter may be omitted, in which case it defaults to the first file name in the input list. The default extension is ".LO" unless the Q option is specified, in which case the default extension is ".MX" (S-record module).

fn3
#
#PR

Error messages, maps, and tables are output to the file/device specified by this parameter. If <fn3> is specified, the output is sent to a disk file with the default extension of ".LL".

If # or #PR is specified, the output is sent to the user's console or the line printer, respectively. If this parameter is omitted, but options requesting output are specified, the listing will be directed to the default output file/device, usually the user's console.

options

The following options are of particular interest to Pascal:

M

This option causes a map of the load module to be generated.

X

This option causes a table of externally defined symbols to be generated.

For a further discussion of the linker and a complete list of options, refer to the M68000 Family Linkage Editor User's Manual.

Examples:

```
LINK  PROG,,#PR;MX
```

The file PROG.RO will be read, the default library PASCALIB will be searched, and a load module PROG.LO will be generated. A load map and table of externally defined symbols will be output to the line printer.

```
LINK  PROG5
```

The file PROG5.RO will be read and the default library PASCALIB will be searched. A load module PROG5.LO will be created and any linker messages will be output to the system default device (usually the user's console).

```
LINK  SUBPROG/PROG,,#PR;MX
```

The file PROG.RO, a Pascal program object module, and the file SUBPROG.RO, a subprogram compiled separately from PROG, will be processed by the linker, and the default library PASCALIB will be searched. A load module, SUBPROG.LO, will be created. A load map and table of externally defined symbols will be output to the line printer.

```
LINK  PROG/SUBPROG,,#PRl;MX
```

The file PROG.RO, a Pascal program object module, and the file SUBPROG.RO, a subprogram compiled separately from PROG, will be processed by the linker, and the default library PASCALIB will be searched. A load module, PROG.LO, will be created. A load map and table of externally defined symbols will be output to the line printer.

5.8 INVOKING THE LINKER TO CREATE A VM01/VERSAbug PROGRAM

Using VERSAdos, a Pascal program can be link-edited and built into an S-record file. Using VERSAbug, the program can then be downloaded into VERSAmodule 01 (see paragraph 7.2 of this manual, and the "System Development" section of the M68KVM01-1, -2 Monoboard Microcomputer User's Guide).

Example:

```
=LINK ,EXAMPLE,EXAMPLE;MSQ
>SEG SEG1(R):8-14 $1000        (Segment 1 is read only, and just after VERSAbug
                                memory.)
>SEG SEG2:0,15                 (Segment 2 is read/write and just after SEG1.)
>IN EXAMPLE                    (Load the program.)
>LIB 0.&.VM01PLIB              (Get VERSAmodule 01 runtime library; PASCALIB is
                                also searched by default.)
>END
=
```

Omitting the parameter <fn1> allows entry of user commands SEG, IN, LIB, and END. The user command IDENT could have been given to provide identifying information for the S0 record. Since it was not used, the defaults are used: the S0 record will use the output file name (EXAMPLE) as the module name, with version and revision numbers of 1.

The file EXAMPLE.RO will be read, and the library file 0.&.VM01PLIB will be searched to resolve external references. SEG1 (read-only sections 8-14), and SEG2 (sections 0 and 15) will be allocated sequentially, starting at address $1000 so that the program EXAMPLE will be loaded past the VERSAbug memory space. SEG1 contains the runtime library and program in sections 8 and 9; SEG2 is a read/write segment containing the device variables and the stack/heap in sections 0 and 15. The S-record file EXAMPLE.MX is created, and the listings produced by the linker are written to the file EXAMPLE.LL.

The options M and S must be specified. The M option generates a load map from which to obtain the address of the symbol START for runtime (paragraph 7.2); the S option provides the proper sequential linking of segments. Option Q specifies that the output file is to be in transportable S-record format.

Before downloading the program EXAMPLE into VERSAmodule 01, the monoboard must be strapped so that there is RAM starting at $1000. The program and its data will then be loaded into RAM starting at location $1000 or above (0-$FFF is reserved for traps and VERSAbug memory).

This method of linking may also be used if the Q option is not specified and the output is a load module which is to be converted to S-record format using the BUILDS utility.

5.9 INVOKING THE LINKER TO CREATE A VERSAdos01/02 PROGRAM

Under VERSAdos on VERSAmodule 01 or 02, a load module with default attributes can be created using the LINK command described in paragraph 5.7, with the following additions/exceptions:

a. If the input file (<fnl>) is entered on the command line, the A option must also be entered.

b. The LIB user command must be entered, as follows, for proper library inclusion:

```
LIB  0.&.PASVMDOS
```

c. There are four non-MMU segments: (1) the data segment is always SEG2 and contains section 15, (2) the code segment contains sections 8 and 9, and is named by the user, (3) a third segment, SEGT, is used temporarily and then deleted during Pascal initialization, and (4) the fourth segment is left for custom applications by the user.

   <start> and/or <end> addresses should not be specified on the SEG command line unless the application requires an absolute address. Rather, VERSAdos should be allowed to decide the appropriate loading address.

d. Users should exercise caution when linking assembly language routines with Pascal routines for VM01, VM02, VMC 68/2, or MVME110. If the routines reference absolute addresses, VERSAdos or other tasks may be inadvertently overwritten, as there is no memory management unit (MMU) to protect the address spaces of various programs from one another.

Example:

```
=LINK  ,EXAMPLE,EXAMPLE;MI
>SEG EXMP(RG):8-14 )
>SEG SEG2:15       )   These two commands could be omitted.
>IN EXAMPLE
>LIB 0.&.PASVMDOS
>END
```

5.10 INVOKING THE LINKER TO CREATE AN RMS68K01/02 PROGRAM

A Pascal program, in order to be executed under RMS68K on VERSAmodule 01 or 02, must first be linked into a boot file by the SYSGEN utility. This linking process is defined within a "chain file" called by SYSGEN through its LINK command. See the example in paragraph 5.10.1.

The "chain file" is not a normal VERSAdos chain file, but is merely input to the linker. The commands in the file are normal linker input, as described in paragraph 5.7, with the following exceptions:

a. If the input file (<fnl>) is entered on the command line, the A option must also be entered.

b. The LIB user command must be entered, as follows, for proper library inclusion:

```
LIB  0.&.PASVMRMS
```

As written, PASVMRMS.RO only supports four devices: #, #CN00, #CN01, and #PR (see paragraph 9.7).

c. There are four non-MMU segments: (1) the data segment is always SEG2 and contains section 15, (2) the code segment contains sections 8 and 9, and is named by the user, (3) a third segment, SEGT, is used temporarily and then deleted during Pascal initialization, and (4) the fourth segment is left for custom applications by the user.

<start> and/or <end> addresses should not be specified on the SEG command line unless the application requires an absolute address. Rather, VERSAdos should be allowed to decide the appropriate loading address.

5.10.1 Invoking SYSGEN To Create A Boot File

The SYSGEN facility is described in the M68000 Family System Generation Facility User's Manual. However, to show the generation of a boot file for a Pascal program, the following example is provided. The example is based on the SORT.SA and FORCE.SA programs listed in Chapter 8.

First, the chain file PASCGEN.CF is called to perform the assembly of FORCE.SA, the compilation of SORT.SA and the call to SYSGEN, as follows:

```
=ASM FORCE
=PASCAL SORT;O
=POPTIM SORT
=PASCAL2 SORT.PO,SORT.RO;L
=DEL PASC.RO;Y
=COPY SORT.RO,PASC.RO
=SYSGEN PASCGEN.SA,/SORTBOOT.SY,SORTBOOT.LS;P
```

The call to SYSGEN contains the argument PASCGEN.SA, which is the input command file. This file contains the following parameters for specifying memory requirements, hardware configuration, and the process/task streams for including load modules in the boot file:

```
*    This file builds up the operating system for a VM2 board system.
*    The operating system includes the EXEC, BIOS, an INITialization
*    task, and a PASCAL task, PASC.
*
*    SYSTEM PARAMETERS
*
UDR=0                          User-defined directive table (none)
MEMEND3=$80000                 End of off-board ram
```

```
GST=4                       Global Segment Table - number of pages
UST=2                       User Semaphore Table - number of pages
TRACE=5                     Trace Table          - number of pages
IOV=1                       I/O Vector Table     - number of pages
MMU=$0                      Address of MMU
TIMER=$F70000               Address of Timer
CLOCKFRQ=800                Number of clock tics per millisecond
TIMINTV=10                  Number of milliseconds between timer interrupts
TIMSLIC=2                   Number of timer interrupts before task forced
*                           to relinguish processor
PANEL=$0                    Front panel address
MEMEND1=$20000              Maximum memory address
MEMEND2=$20000
TRCFLAG=$C000               Trace flag
WHERLOAD=$0                 Memory address where boot file will be loaded
PAT=2                       Number of pages in the periodic activation table
BUGTRAC=$F000BC             Address of VERSAbug's trace routine
PC=$E00                     Initialize Program Counter
STACK=$C00                  Stack location
KILVECT=142                 Killer vector number
SERPTS=140                  Serial port vector number
PTMVECT=28                  Timer vector number
FAIL=141                    AC fail vector number
SWABRT=31                   Software abort vector number
NRAD1=0                     Number of RAD1 boards on system
DPRVAO=0                    Dual ported RAM VERSAbus addr offset
NUSRRAD=0                   Number of RAD1 users/boards
IOBINT4=$74                 I/O channel interrupt vector #
IOBINT3=$73                 I/O channel interrupt vector #
IOBINT2=$72                 I/O channel interrupt vector #
IOBINT1=$71                 I/O channel interrupt vector #
BCLRV=147                   Bus clear interrupt vector #
*
*     Build EXEC
*
STARTRMS=$F00
PROCESS VM2.RMSV2.LO
END EXEC
MSG EXEC BUILT
*
*     Build BIOS
*
MEMBEG=*
TASK VM2.BIOS.LO
BIOSSTRT=*
SUBS VM2.LBIOS.CF
LINK VM2.LBIOS.CF
SESSION=1
PRIORITY=200
END BIOS
MSG BIOS BUILT
*
*   Build PASCAL program
*
TASK VM2.PASC.LO
PASCSTRT=*
```

```
SUBS VM2.PASC.CF
LINK VM2.PASC.CF
SESSION=2
PRIORITY=100
ATTRIB='USER'
END PASC
MSG PASC BUILT
*
*   Build INIT
*
PROCESS VM2.INIT.LO
SUBS VM2.INTIOV2.SA
ASM VM2.EQUTIMER.SA/VM2.INTIOV2.SA,VM2.INTIOV2.RO,VM2.INTIOV2.LS;Z=100
SUBS VM2.INDV.SA
ASM VM2.INDV,VM2.INDV,VM2.INDV;Z=100
INTSTR=*
SUBS VM2.LNKINT2.CF
LINK VM2.LNKINT2.CF
END INIT
MSG INIT BUILT
END
```

The first TASK stream reads the chain file, VM2.LBIOS.CF. This file contains the following linkage editor commands to link BIOS into the boot file:

```
=LINK ,VM2.BIOS.LO,VM2.BIOS.LL;MIX
SEG SEG0:8 \BIOSSTRT
IN VM2.BIOS
END
```

The second TASK stream reads the chain file, VM2.PASC.CF. This file contains the following linkage editor commands to link the programs PASC.RO (SORT.RO) and FORCE.RO into the boot file:

```
=LINK ,VM2.PASC,VM2.PASC;IMS
SEG CODE(RG):8-14 \PASCSTRT
SEG SEG2:15
IN VM2.PASC,VM2.FORCE
LIB 0.&.PASVMRMS.RO
END
```

CAUTION

BECAUSE THE PASCAL PROGRAM CONTAINS TWO SEGMENTS, THE USER MUST EITHER SPECIFY THE S OPTION ON THE LINKER COMMAND LINE TO ALLOCATE THE SEGMENTS SEQUENTIALLY OR EXPLICITLY SPECIFY THE STARTING ADDRESS OF SEG2.

The final PROCESS stream reads the chain file, LNKINT2.CF. This file contains the following linkage editor commands to link the initialization routine into the boot file:

```
=LINK ,VM2.INIT.LO,VM2.INIT.LL;IXHM
SEGMENT .INT:8 \INTSTR
INPUT VM2.INIT.RO,VM2.INTIOV2.RO,VM2.INDV.RO,VM2.SYSPARV.RO
END
=END
```

The following output listing, found in the file SORTBOOT.LS, contains the results of the sysgen process:

```
COMMAND LINE PARAMETERS:  SYSGEN.SA,/SORTBOOT.SY,SORTBOOT.LS;P


. *
. *     THIS FILE BUILDS UP THE OPERATING SYSTEM FOR A VM2 BOARD SYSTEM.
. *     THE OPERATING SYSTEM INCLUDES THE EXEC, BIOS, AN INITIALIZATION
. *     TASK, AND A PASCAL TASK, PASC.
. *
. *     SYSTEM PARAMETERS
. *
. UDR=0                          USER-DEFINED DIRECTIVE TABLE (NONE)
. MEMEND3=$80000                 END OF OFF-BOARD RAM
. GST=4                          GLOBAL SEGMENT TABLE - NUMBER OF PAGES
. UST=2                          USER SEMAPHORE TABLE - NUMBER OF PAGES
. TRACE=5                        TRACE TABLE          - NUMBER OF PAGES
. IOV=1                          I/O VECTOR TABLE     - NUMBER OF PAGES
. MMU=$0                         ADDRESS OF MMU
. TIMER=$F70000                  ADDRESS OF TIMER
. CLOCKFRQ=800                   NUMBER OF CLOCK TICS PER MILLISECOND
. TIMINTV=10                     NUMBER OF MILLISECONDS BETWEEN TIMER INTERRUPTS
. TIMSLIC=2                      NUMBER OF TIMER INTERRUPTS BEFORE TASK FORCED
. *                              TO RELINQUISH PROCESSOR
. PANEL=$0                       FRONT PANEL ADDRESS
. MEMEND1=$20000                 MAXIMUM MEMORY ADDRESS
. MEMEND2=$20000
. TRCFLAG=$C000                  TRACE FLAG
. WHERLOAD=$0                    MEMORY ADDRESS WHERE BOOT FILE WILL BE LOADED
. PAT=2                          NUMBER OF PAGES IN THE PERIODIC ACTIVATION TABLE
. BUGTRAC=$F000BC                ADDRESS OF VERSABUG'S TRACE ROUTINE
. PC=$E00                        INITIALIZE PROGRAM COUNTER
- PC = $000E00
. STACK=$C00                     STACK LOCATION
. KILVECT=142                    KILLER VECTOR NUMBER
. SERPTS=140                     SERIAL PORT VECTOR NUMBER
. PTMVECT=28                     TIMER VECTOR NUMBER
. FAIL=141                       AC FAIL VECTOR NUMBER
. SWABRT=31                      SOFTWARE ABORT VECTOR NUMBER
. NRAD1=0                        NUMBER OF RAD1 BOARDS ON SYSTEM
. DPRVAO=0                       DUAL PORTED RAM VERSABUS ADDR OFFSET
. NUSRRAD=0                      NUMBER OF RAD1 USERS/BOARDS
. IOBINT4=$74                    I/O CHANNEL INTERRUPT VECTOR #
. IOBINT3=$73                    I/O CHANNEL INTERRUPT VECTOR #
. IOBINT2=$72                    I/O CHANNEL INTERRUPT VECTOR #
. IOBINT1=$71                    I/O CHANNEL INTERRUPT VECTOR #
. BCLRV=147                      BUS CLEAR INTERRUPT VECTOR #
. *
. *      BUILD EXEC
. *
. STARTRMS=$F00
. PROCESS VM2.RMSV2.LO
. END EXEC
- PC = $005200
```

```
. MSG EXEC BUILT
. *
. *     BUILD BIOS
. *
. MEMBEG=*
. TASK VM2.BIOS.LO
. BIOSSTRT=*
. SUBS VM2.LBIOS.CF
. LINK VM2.LBIOS.CF
- =LINK ,VM2.BIOS.LO,VM2.BIOS.LL;MIX
. SESSION=1
. PRIORITY=200
. END BIOS
- PC = $005F00
. MSG BIOS BUILT
. *
. *   BUILD PASCAL PROGRAM
. *
. TASK VM2.PASC.LO
. PASCSTRT=*
. SUBS VM2.PASC.CF
. LINK VM2.PASC.CF
- =LINK ,VM2.PASC,VM2.PASC;IMS
. SESSION=2
. PRIORITY=100
. ATTRIB='USER'
. END PASC
- PC = $00DB00
. MSG PASC BUILT
. *
. *   BUILD INIT
. *
. PROCESS VM2.INIT.LO
. SUBS VM2.INTIOV2.SA
. ASM VM2.EQUTIMER.SA/VM2.INTIOV2.SA,VM2.INTIOV2.RO,VM2.INTIOV2.LS;Z=100
- =ASM  VM2.EQUTIMER.SA/VM2.XINTIOV2.SA,VM2.INTIOV2.RO,VM2.INTIOV2.LS;Z=100
. SUBS VM2.INDV.SA
. ASM VM2.INDV,VM2.INDV,VM2.INDV;Z=100
- =ASM  VM2.XINDV,VM2.INDV,VM2.INDV;Z=100
. INTSTR=*
. SUBS VM2.LNKINT2.CF
. LINK VM2.LNKINT2.CF
- =LINK ,VM2.INIT.LO,VM2.INIT.LL;IXHM
. END INIT
- PC = $00E300
. MSG INIT BUILT
. END

FILE NAME     TASK  PROC  SEG   ADDR      TCB

RMSV2.LO            RMSV  RMS0  $000E00
                          RMS2  $000F00
BIOS.LO       BIOS        SEG0  $005200   $005D00
PASC.LO       PASC        CODE  $005F00   $00D900
                          SEG2  $007600
INIT.LO             INIT  .INT  $00DB00
```

```
- FINAL PC VALUE = $00E300
- SYSGEN DONE--START-UP ADDRESS IS $00DB00

0  ERRORS ENCOUNTERED
```

CHAPTER 6

ASSEMBLY ROUTINE LINKAGE

6.1 GENERAL

An assembly language routine may be called externally by a Pascal program using normal Pascal argument passing. Such a routine may:

a. Perform a function not available in Pascal -- e.g., data manipulation or I/O not provided in the runtime library, or some mathematics not supported by Pascal.

b. Optimize some code to be used repetitively in a real-time environment. The Pascal compiler does optimize, but a user-written assembly language routine may be shorter and faster.

6.2 PROGRAM PREPARATION

There are two requirements which must be satisfied in order to include an assembly language subroutine in a Pascal program. The first is to declare the external assembly language routine in the Pascal program. This is done by declaring a level 1 procedure or function, contained in the main program or a subprogram, using the forward directive. A good place for these declarations is prior to the first non-external procedure heading.

For example:

```
FUNCTION   SUMTHREE(I,J,K:INTEGER):INTEGER; FORWARD;
```

The external assembly language subroutine may then be called just as any Pascal procedure or function.

The second requirement concerns the file which contains the assembly language routine. This file must have an entry point, which has been declared external with an XDEF, with the same name (truncated to 8 characters) as the procedure or function in the Pascal program. The assembler must be informed that the subroutine is to be included in section 9. A 'SECTION 9' directive at the beginning of the assembly language subroutine file accomplishes this.

6.3 CALLING A ROUTINE

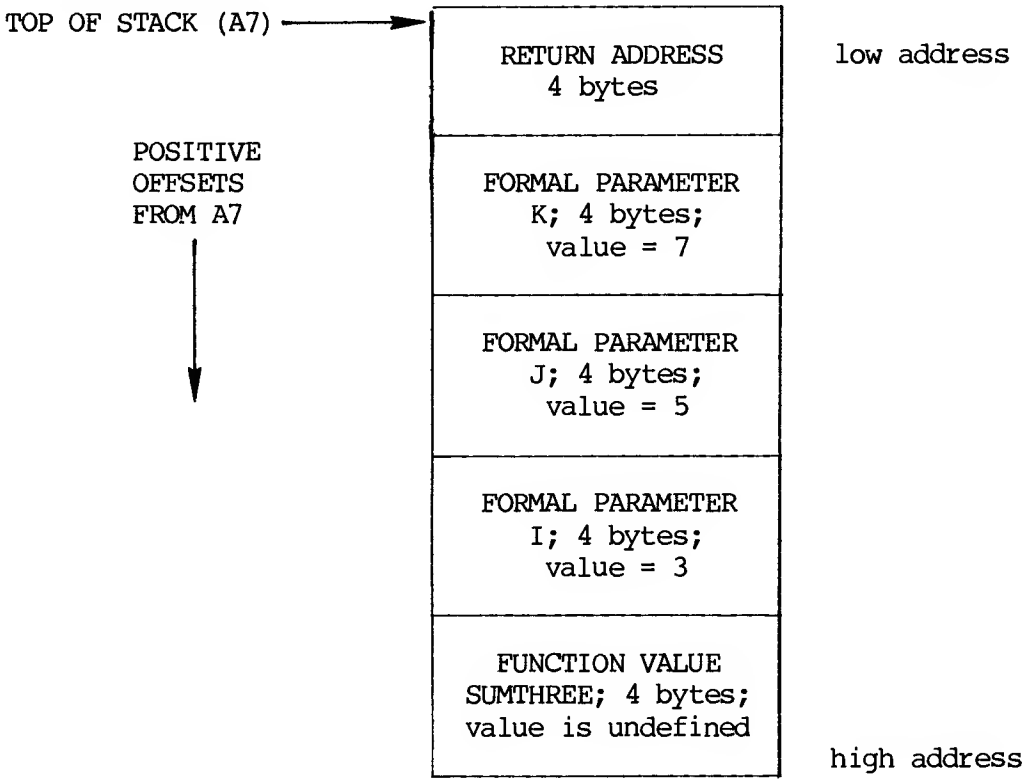
Calling an assembly language routine is identical in format -- and its runtime requirements are identical in system usage -- to a regular function or procedure call in Pascal. Parameters, for example, are placed on the top of the stack, beneath the return address, in the order they are declared -- the first parameter is stacked first and the last parameter is nearest the top of the stack. If the assembly language routine is declared a function, the space for the return value is below the first parameter on the stack (i.e., the address contained in A7 plus a positive displacement).

For example, given the declaration and call in the following Pascal program fragment:

```
FUNCTION   SUMTHREE(I,J,K:INTEGER):INTEGER; FORWARD;


BEGIN
  A:= SUMTHREE(3,5,7);
```

the stack would look as follows upon entry to the assembly language subroutine named SUMTHREE:



The size of parameters depends on the type.

A VAR parameter passes a four-byte address of the actual parameter, which can be used to reference the actual parameter via indirection. A value parameter passes the value of the expression which corresponds to the formal parameter.

Boolean parameters occupy two bytes on the stack, but only the byte closer to the top of the stack contains valid data. This byte has the value of one for true and the value of zero for false.

Character parameters use two bytes on the stack, but only the byte closest to the top of the stack contains valid data. This byte has the value of the ASCII code for the character passed in it.

Integer parameters occupy four bytes on the stack. They are stored as 32-bit two's complement numbers. Integer subrange types that fall into the range -128..127, inclusive, use two bytes on the stack, but only the byte closer to the top of the stack contains valid data. They are stored as 8-bit two's complement numbers. Integer subrange types that extend outside of the range -128..127, inclusive, but are within the range -32768..32767, inclusive, use two bytes on the stack. They are stored as 16-bit two's complement numbers.

Real parameters occupy four bytes on the stack, with the sign bit being closest to the top of the stack. Dreal parameters occupy eight bytes on the stack, with the sign bit being closest to the top of the stack. Xreal parameters occupy ten bytes on the stack, with the sign bit being closest to the top of the stack. The internal representation of real, dreal, and xreal values is described in Chapter 11.

Set parameters require eight bytes on the stack, with the byte nearest the top of the stack containing bits 63-56 and the byte farthest from the top of the stack containing bits 7-0.

Arrays and records occupy a number of bytes equal to their length, plus one if they are of an odd length. The filler byte is the byte farthest from the top of the stack.

Strings should always be passed to assembly language routines as VAR parameters, due to the complexity of determining their actual size on the stack.

Pointers require four bytes on the stack and they contain the address of the variable they reference.

The assembly language subroutine is responsible for preserving the value of registers A3, A5, and A6 during its execution. It is also responsible for removing from the stack all parameters passed to it by the Pascal program, and for storing a value in the return value location on the stack if the subroutine was declared as a function.

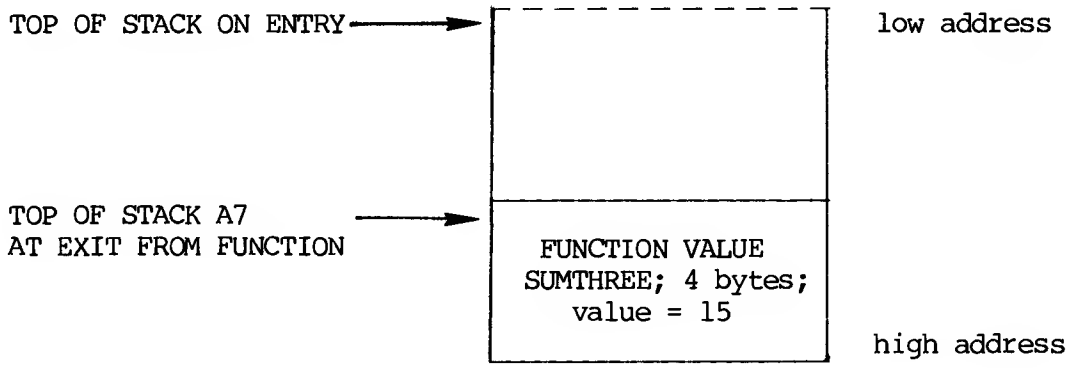
The values of the A5 and A6 registers may be of use to the assembly language routine, since A5 points to the base of the global variable area and A6 points to the base of the local variable area of the procedure or function which was being executed when the assembly language routine was called. To reference a variable in either of these areas, a negative displacement from the register must be used.

The assembly language subroutine is free to use the space between the top of the stack (pointed to by A7) and the top of the heap for local data storage. The address of the top of the heap is kept in the long word which is located in memory at a positive offset of four from the address in register A5 (see Table 9-10).

If A7 ever contains an address that is less than the address of the top of the heap, a stack/heap overflow condition has occurred. If a stack/heap overflow has occurred, then both the stack and the heap may contain invalid data.

Control may be returned to the Pascal program by means of either a return from subroutine instruction or a jump indirect through an address register which contains the return address. No matter which method is used, it is up to the assembly language subroutine to adjust the stack so as to remove the passed parameters. If the assembly language routine returned a function value, then A7 should point to that location on the stack where the space was reserved for the return value prior to assembly language routine entry. If the assembly language routine did not return a function value, A7 should point just below where the first parameter was pushed on the stack.

Following is a picture of the stack for the SUMTHREE routine, seen earlier, just before the return to the Pascal program:



6.4 ASSEMBLY ROUTINE LINKAGE

An assembly language routine is linked with a Pascal program by means of the linkage editor (Chapter 5). An example is shown in Chapter 8.

CHAPTER 7

RUNNING A PASCAL PROGRAM

7.1 RUNNING A PROGRAM UNDER VERSAdos

Under VERSAdos (on EXORmacs, VME/10, VM01, VM02, or MVME110), the linkage editor creates a runnable program whose execution may be started by merely typing its name.

7.1.1 Runtime File Assignment

File assignments may be made in three ways, as described in the following paragraphs: in command lines, in reset/rewrite statements, and as assignments passed to the program.

7.1.1.1 Command Line Assignment. If the Pascal program needs any external files, their names may be passed on the command line. The files are associated one-for-one with the file identifiers in the program statement of the executing program, with the exception that the two file identifiers, input and output, are ignored.

Therefore, given the following program statement in a Pascal program, where the name of the program is "compute":

```
program   compute(source,object,listing);
```

and given the command line to VERSAdos:

```
COMPUTE   MATH.SA,TRIG.SA,ARITH.LS
```

the file identifiers are associated as follows with the command line files:

```
source = MATH.SA
object = TRIG.SA
listing = ARITH.LS
```

Pascal standard input and output files are special in that they must be preceded in the command line with I= for input, and O= for output. This means that for standard input and output, the command line order is not important. In most cases, input and output will not need to be specified, since input defaults to the command initiating file or device, and output defaults to the log/error file or device. Both of these are the user terminal in the interactive mode.

Therefore, modifying the above example program statement to

```
program   compute(input,output,source,object,listing);
```

does not change the meaning of the example command line above. Some equivalent examples are:

```
COMPUTE   I=#,O=#,MATH.SA,TRIG.SA,ARITH.LS
COMPUTE   I=#,MATH.SA,TRIG.SA,ARITH.LS
COMPUTE   I=#,MATH.SA,O=#,TRIG.SA,ARITH.LS
```

The device name '#' means the user's terminal. To change the device so that I/O to standard file output goes to the printer, the command line would look like

```
COMPUTE   I=#,MATH.SA,TRIG.SA,ARITH.LS,O=#PR
```

or

```
COMPUTE   O=#PR,MATH.SA,TRIG.SA,ARITH.LS
```

If an extension is not specified on the command line, it will default to .SA for a user program, so the following is again equivalent to the last example above:

```
COMPUTE   MATH,TRIG,ARITH.LS,O=#PR
```

If a file is left off the command line, it will default to a temporary/local file that will be deleted at the end of the program. This default may be overridden during execution by runtime file assignment.

Logical concatenation of input is supported in the command line -- that is to say, if you have two or more input files, the Pascal program may view them as one logical input stream, as in the following:

```
COMPUTE   MATH/MATH1,TRIG,ARITH.LS
```

where, when all the data from MATH is exhausted, data will be read from MATH1; and if "source" (see foregoing program statement) is reset, such as in a two-pass compilation, data will be read starting from MATH again. The end-of-file status will be true only at the end of the last file in the file list -- MATH1 in the example above.

7.1.1.2 Reset/Rewrite File Assignment. File assignment using the reset and rewrite statements is described in Pascal Programming Structures for Motorola Microprocessors. The reset and rewrite statements take the following forms:

```
reset(<logical file name>,<string>)

rewrite(<logical file name>,<string>)
```

The <logical file name> is an identifier that must be predeclared in a program var statement (with the exception of standard identifiers input and output). The <string> may be any string-valued expression, including but not limited to constants and variables. The value of <string> is the resource name string (RNS) and is defined as follows, using Backus-Naur Form (paragraph 1.7).

Resource Name String (RNS):

In this definition of the RNS, any variables not defined are found in Appendix A of Pascal Programming Structures for Motorola Microprocessors.

```
<resource name string> ::= <file descriptor>[;<option>[,<option>]...]|
     ;<option>[,<option>]...
<file descriptor> ::= <device name>|
     <volume name>:[<user number>.<catalog>.<file name>.<extension>[(<key>)]]|
     <user number >.<catalog>.<file name>.<extension>[(<key>)]|
     <catalog>.<file name>.<extension>[(<key>)]|
     <file name>[.<extension>[(<key>)]]|
     .<extension>[(<key>)]
<device name> ::= #<device mnemonic>
<device mnemonic> ::= <console>|PR[<digit>]|NULL|<empty>|<identifier>
<console> ::= CN<digit><digit>
<digit> ::= 0|1|2|3|4|5|6|7|8|9
<volume name> ::= <identifier>|<empty>
<user number> ::= <digit sequence>|<empty>
<catalog> ::= <identifier>|<empty>
<file name> ::= <identifier>|<empty>
<extension> ::= <identifier>|<empty>
<key> ::= [<key letter><key letter>]<key letter><key letter>
<key letter> ::= A|B|C|D|E|F|G|H|I|J|K|L|M|N|O|P
<option> ::= B|C=<digit sequence>|D=<digit sequence>|
     F=<digit sequence>|[-]R|[-]W
```

<device mnemonic> is a maximum of four characters. An <empty> device mnemonic specifies the user's terminal; or other device <identifier>'s defined in the system may be specified.

<volume name> accepts a maximum of four alphanumeric characters; <user number> is in the decimal range 0-9999; <catalog> and <file name> each accept a maximum of eight alphanumeric characters; <extension> is a field of two alphanumeric characters. These fields are described in the VERSAdos System Facilities Reference Manual.

<key> specifies the fixed protection for the file. Two or four characters, in the range A through P, may be entered as the key. The rightmost two characters are the read code. The leftmost two characters are the write code.

A code of PP permits public access for the read or write function to which the code applies. A code of OO permits only owner and supervisor access. Other codes would be any combination of two <key letter>s. If two characters are entered, the system defaults to PP to the left of the entered characters. For example, a key of AC is the same as PPAC. Likewise, a key of PF is equivalent to PPPF. If no key is specified, default is PPPP. When a file is allocated (created), its protection codes are saved with the file. After that, the appropriate code must be matched to access (assign) the file. The VERSAdos Data Management Services and Program Loader manual describes the complete use of fixed protection codes.

The <resource name string> can be used to specify key and options for a file specified on the command line. However, if any portion of the file descriptor other than the key is specified in the resource name string, the system defaults for the fields not specified in the resource name string will override the corresponding fields in the command line. If a resource name string containing a file descriptor is used in a reset or rewrite of a local file, the file is then considered to be external -- that is, the file will not be deleted automatically when it is closed.

The <option> field is preceded by a semicolon. If more than one option is specified, equals options (C, D, and F) must have a comma separating the digit sequence from any following option; other options may have no separation or be separated by a comma. The following options are permitted:

B	Binary data type. (ASCII data type is the default for text files. Binary data type is the default for all other files.)
C=<digit sequence>	Contiguous file type, file size in sectors. (Sequential file type is the default.) A sector contains 256 bytes.
D=<digit sequence>	Sequential file type, data block size in sectors. Number must be 0 or in the range 4-255. (Default is 0, which VERSAdos interprets as four sectors.) The data block size is the number of sectors per data storage block on the disk. A sector contains 256 bytes. In order to perform I/O with file components greater than 1K (four sectors), a data block large enough to accommodate a component must be specified.
F=<digit sequence>	Sequential file type, File Access Block (FAB) size in sectors. Number must be in the range of 0-20. (Default is 0, which VERSAdos interprets as one sector.) The FAB contains pointers to the disk data storage blocks of the file. A sector contains 256 bytes.
R	Public read access permission.
-R	Exclusive read access permission.
W	Public write access permission.
-W	Exclusive write access permission.

The permissible option combinations of the access permissions are: R, -R, W, -W, RW, -RW, R-W, and -R-W. The default access permissions are R for reset and W for rewrite. If the file is a temporary file, the default access permissions are R-W.

Entering any access permission overrides the default permission for the current and any subsequent resets and rewrites. Therefore, some type of permission should always accommodate the operations on the file. For example, entering -R for a file to be written to will generate an error because there is no write access permission. In this example, -RW or -R-W should have been entered to gain write or exclusive write permission while also retaining exclusive read permission. The VERSAdos Data Management Services and Program Loader manual contains a complete description of access permission.

Resource Name String Examples:

```
rewrite(sortedfile,'.(CCDD);RW');
```

For "sortedfile" in program statement:

If the command line specifies a file assignment for "sortedfile", then the rewrite statement will open the file for output, using the file descriptor specified in the command line, but with a key of CCDD and with both read and write file access permission. (If write-read codes are specified on the command line, they will be overridden by the key in the resource name string.)

If the command line does not specify a file assignment, then a temporary file will be opened for output with a key of CCDD and with both read and write file access permission. The temporary file will be automatically deleted when the program terminates.

For "sortedfile" not in the program statement:

A temporary file will be opened for output with a key of CCDD and with both read and write file access permission. The temporary file will be automatically deleted when the program terminates.

```
rewrite(errorlogfile,'LOG1:.ERRORS.LEDGER.LG(AB);-W');
```

Whether or not "errorlogfile" is in the program statement, the file assignment for "errorlogfile" will be as specified in the rewrite statement. The volume name will be LOG1; the default user number will be used; the catalog, file name, and extension will be ERRORS.LEDGER.LG; the file will have a key of PPAB and exclusive write access permission.

```
reset(initfile,'TANGENT.DT');
```

The Pascal file "initfile" will be assigned to the VERSAdos file TANGENT.DT. The default values for volume name, user number, catalog, and keys will be used. This will happen even if "initfile" is included in the program statement and is then specified on the command line as some other file or device.

```
rewrite(listfile,'#LP');
```

The Pascal file "listfile" will be assigned to the device identified as LP. This will happen even if "listfile" is included in the program statement and is then specified on the command line as some other file or device.

Alternatively, a string constant could be replaced by an identifier of string type that has been assigned the value of the <resource name string>. For example:

```
var device: string [80];
      :
begin device:= '#LP';
      :
  rewrite(listfile,device);
```

7.1.1.3 Passed Assigned Files. A file may be passed assigned to a Pascal program when it begins execution. This is accomplished by assigning the file to the appropriate logical unit number using the VERSAdos ASSIGN command before executing the Pascal program. The appropriate logical unit number is the file identifier's position (starting with 1) in the PROGRAM statement, ignoring the identifier's input and output. (Input is logical unit number 5, and output is logical unit number 6.) File name specifications on the command line or in resource name strings in reset or rewrite statements override passed file assignments. Refer also to paragraphs 9.3 and 9.5.

7.1.1.4 Default Values for File Descriptor.

<volume name>	The default user volume is supplied by VERSAdos when the user number is non-zero. If a user number of 0 is entered without a volume name, the system default volume is used, unless the logon user number was 0 and the VERSAdos 'USE' command had been used to specify a default user number. In this case, the default volume name will be the default user volume. If the file name starts with & (temporary file) and no volume name is specified, the user default volume is used. If the file starts with @ (spooler file) and no volume name is specified, then the system default spooler volume will be used. These other default volume names are also supplied by VERSAdos.
<user number>	The user number specified in the log on or in the last USE command is the default.
<catalog>	The catalog specified in the log on or in the last USE command.
<file name>	There is no default file name. The file name must always be specified, except for temporary/local files for which VERSAdos generates unique file names.
<extension>	The default extension is SA.
<key>	The default key is PPPP.
<data type>	The default data type is ASCII for text files and binary for nontext files.
<file type>	The default file type is sequential. For a sequential file, the default data block size is four sectors and the default file access block size is one sector. The default record length is variable for a sequential text file. For a sequential file that is not a text file, the default record length is equal to the component size.
<access permission>	The default access permission is public read for reset and public write for rewrite. However, temporary files have a default access permission of public read and exclusive write for both reset and rewrite.

7.1.2 Stack/Heap Memory Segment

If the Pascal program does not run with the default stack/heap memory segment size, it may be extended at runtime with the Z option. Suppose, for example, the program COMPUTE was allocated 16K (16,384) bytes for the stack/heap segment by Phase 1 (via default calculations or user-specified H and/or S source program options (see Table 2-1)), but required more stack/heap space in order to execute. The command line

```
COMPUTE   MATH.SA,TRIG.SA,ARITH.LS;Z=32
```

would make the stack/heap memory segment 32K (32,768) bytes in length. The number following the 'Z=' is always the number of K (where one K is 1024 bytes). The Z option will not reduce the stack/heap size below the size calculated by Phase 1.

7.2 RUNNING A PROGRAM ON VM01 with VERSAbug

After system setup (refer to the VERSAbug Debugging Package User's Manual, and "System Setups" in the M68KVM01-1, -2 Monoboard Microcomputer User's Guide), a Pascal program may be run on VERSAmodule 01. The following example procedure assumes a VERSAmodule 01 whose serial port 1 is connected to an EXORterm 155 control terminal, and whose serial port 2 is connected to an EXORmacs Development System:

a. Apply power to VERSAmodule 01. If the board is set up and operating correctly, the HALT and BRDFAIL LED's will light up momentarily, then go out, and the console will display:

```
VERSAbug 1.0 >
```

b. If step 1 (powering up) was performed previously, or if "VERSAbug 1.0 >" is not displayed, press the RESET button on the front edge of the monoboard.

c. Initialize VERSAbug memory with a BI or BT command and then enter transparent mode by typing:

```
TM
```

d. Log on to VERSAdos as follows:

   a. Press the BREAK key,
   b. Enter the user number,
   c. Optionally enter the security word and password, as required.

e. Exit transparent mode by typing:

```
CTRL-A
```

(that is, while holding down the CTRL key, press the A key.) The console should display:

```
VERSAbug 1.0 >
```

f. Download the file of S-records (paragraph 5.8) into VERSAmodule memory by typing:

    LO ;X =COPY <file>,#

    where <file> is the VERSAdos name of the user input file. As the file is being downloaded, the S-records -- as well as any messages that occur -- are displayed on the console.

g. Set the User Stack Pointer and the Supervisor Stack Pointer to appropriate values in high memory.

h. Type the command:

    GO <start>

    where <start>, a memory address, is the value of the symbol START defined in the module TABLE. TABLE and START are found in the load map listing, generated by specifying option M on the LINK command line.

7.2.1 Use of the Resource Name String

On VERSAmodule 01, there is no command line in which to specify a file name. Rather, a <resource name string> may be used -- as a parameter in a reset or rewrite statement (paragraph 7.1.1) -- to specify a device name, in accord with the following BNF syntax:

<resource name string> ::= <device name>

<device name> ::= # | #PR | #PRT1 | #PRT2

or #PRT1 specifies the terminal port for standard input/output; #PR specifies the line printer as the output device. For the standard files "input" and "output", the default device name is #PRT1 (the terminal port); for all other files, the default is #PRT2 (the host port).

7.3 RUNNING A PROGRAM ON VERSAmodule 01, 02, or MVME110 UNDER RMS68K

After running SYSGEN (see paragraph 5.10), the boot file -- containing RMS68K, BIOS, the initialization task, and the user's Pascal program -- may be loaded and executed on a VERSAmodule 01 or 02 by executing a bootstrap (BO) command.

Under VERSAbug, enter the BO command in the format (the format is described in the VERSAbug Debugging Package User's Manual):

```
BO  [<device>,<controller>,<string>]
```

<string> is the file descriptor of the boot file, for example:

```
BO  0,1,45.CIRF.PRNT.SY
```

The BO command normally calls the IPL.SY task, pointed to by sector 0 of the disk(ette). IPL, in turn, loads the file specified by <string>.

The <string> parameter, however, may be eliminated from the command line if sector 0 has been previously changed to point directly to the boot file. This change can be done as follows:

a. Log on to VERSAdos under user number 0,

b. Enter a DIR utility command of the form:

   DIR <boot file>;A

   where <boot file> is the descriptor of the file created by SYSGEN.

   Save the values of START (xxxx) and SIZE (yyyy) from this display for use in the DUMP command below (step 3).

c. Enter a DUMP utility command to modify sector 0 of the device on which the boot file is found. At offset $14 of sector 0, enter a 4-byte starting sector address ($xxxxxxxx + 1); at offset $18, enter a 2-byte sector count ($yyyy -1); and at offset $1E, enter a 4-byte load address ($zzzzzzzz), which is the value associated with the first "PC" command given to SYSGEN when generating the boot file.

Example:

```
=DIR PRINTER.SY;A

DIR VERSION 111781 3   7/ 6/82 15:21:38

FIX:0000..PRINTER.SY
                  LOG   # OF                          REC  KEY FAB DB   DATE      DATE
  START    END    EOF  RECORDS  WC  RC  FT    LEN  LEN LEN LEN CHANGED  ACCESSED
 $15CEB $15D76    -      -      PP  PP   C     -    -   -   -   6/28/82   6/28/82

  SIZE       140/$8C

TOTAL SIZE          140/$8C

NUMBER FILES RETRIEVED =      1
```

```
=DUMP FIX;I
DUMP VERSION  011882 3

> R 0
> M 14
14    00   '.'   00
15    00   '.'   01      This is the START sector address +1.
16    00   '.'   5C
17    29   ')'   EC
18    00   '.'   00      This is the SIZE sector count -1.
19    1D   '.'   8B
1A    00   '.'   .
> M 1E
1E    03   '.'   00      This is the load address -- the value associated
1F    34   '4'   00      with the first "PC command input to SYSGEN when
20    00   '.'   0E      generating the boot file.
21    00   '.'   00
22    00   '.'   .
> W
> Q
=
```

7.3.1 Use of the Resource Name String

Under RMS68K, there is no command line on which to specify a file name. Rather, a <resource name string> may be used -- as a parameter in a reset or rewrite statement (paragraph 7.1.1) -- to specify a device name, in accord with the following BNF syntax:

```
<resource name string> ::= <device name>
<device name> ::= # | #CN00 | #CN01 | #PR
```

or #CN00 specifies the terminal port for standard input/output; #CN01 specifies a second terminal; and #PR specifies a line printer as the output device. For the standard files "input" and "output", the default device name is #CN00; for other files, there is no default.

Note that Pascal tasks which do I/O to the above devices must use BIOS in their application system.

7.3.2 Debugging Pascal Tasks Under RMS68K

The general problem of testing and debugging tasks in an application system based upon RMS68K is taken up in Chapter 5 of the M68000 Family Real-Time Multitasking Software User's Manual. The user should be familiar with this material before attempting to build application systems containing Pascal tasks.

Chapter 5, referred to above, promotes the use of an exception monitor as a debug aid and for monitoring alarm situations. This mechanism is ideal for debugging problem Pascal tasks. All failing Pascal tasks first issue a TRAP #4 instruction with D0 containing the runtime error number. The exception monitor can capture this TRAP #4 and print out debug information at this point. Alternatively, the Pascal task will continue execution by aborting to its calling monitor with the RMS68K ABORT directive (this is a TRAP #1 with D0=14). Debug information can also be displayed at this point by the monitor which called the Pascal task. In either of these two cases, the relevant debug information is as follows:

offset $170 from A5 - contains the program counter address immediately following the instruction which failed (four bytes).

offset $174 from A5 - contains the runtime error number which occurred (four bytes).

offset $178 from A5 - contains a dump of the registers when the runtime error occurred (16 times 4 = 64 bytes).

This information should point the user to the exact line in the Pascal task where the runtime error occurred.

CHAPTER 8

SAMPLE PROGRAMS

8.1 PROGRAM FOR VERSAdos EXECUTION

8.1.1 Phase 1 Listing

```
Line      Loc Lev BE Motorola Pascal        SORT    .SA

   1(       0) 0)— {-------------------------------------------------------------}
   2(       0) 0)— {                                                             }
   3(       0) 0)— {                          S O R T                            }
   4(       0) 0)— {                                                             }
   5(       0) 0)— {        This program demonstrates the linking of a Pascal program  }
   6(       0) 0)— {        with an assembly language subroutine.                }
   7(       0) 0)— {                                                             }
   8(       0) 0)— {        The driver program simply asks for an array of numbers,  }
   9(       0) 0)— {        one by one, sorts the numbers in increasing numerical  }
  10(       0) 0)— {        order, and prints the results.                       }
  11(       0) 0)— {                                                             }
  12(       0) 0)— {        An assembly language routine is used to force output  }
  13(       0) 0)— {        to a text file without having to do a writeln.  This  }
  14(       0) 0)— {        allows for prompting for input and having the input  }
  15(       0) 0)— {        entered on the same line.                            }
  16(       0) 0)— {                                                             }
  17(       0) 0)— {-------------------------------------------------------------}
  18(       0) 0)—
  19(     -16) 0)— PROGRAM  sort (input,output);
  20(     -16) 0)—
  21(     -16) 0)—    CONST
  22(     -16) 0)—       max_array_size  = 5000;                  {maximum array size}
  23(     -16) 0)—
  24(     -16) 0)—    TYPE
  25(     -16) 0)—       index_range     = 1..max_array_size;     {range of indices into array}
  26(     -16) 0)—
  27(     -16) 0)—    VAR
  28(  -20016) 0)—       number_array:    ARRAY [index_range] of integer;    {the array}
  29(  -20018) 0)—       array_size:      0..max_array_size;       {actual size of array}
  30(  -20022) 0)—       i,j:             index_range;             {indices into array}
  31(  -20026) 0)—       temp:            integer;                 {used for swaping elements}
  32(  -20027) 0)—       exchange:        boolean;                 {any exchanges in last pass?}
  33(  -20027) 0)—
  34(  -20027) 0)—    {declare the needed assembly language routine as external}
  35(  -20027) 0)—
  36(       0) 1)—    PROCEDURE  force (VAR fil: text);  FORWARD;
  37(       0) 1)—
  38(       0) 1)—    {$A=2}
  39(       0) 1)—
  40(       0) 1)—    {the main program starts here}
  41(       0) 1)—
**** FORCE    Assumed external
  42         1  0)A-   BEGIN {sort}
  43            0)—
  44            0)B-      REPEAT                                  {loop for each array}
  45            0)—
  46            0)—          {ask for size of the array}
  47            0)—
  48         2  0)—          writeln (output);   writeln(output);
  49         4  0)—          write (output,'Input size of array (0 to quit): ');
  50         5  0)—          force (output);
  51            0)—
  52         6  0)—          readln (input,array_size);              {get size of array}
  53            0)—
  54         7  0)—          IF array_size > 0 THEN
  55            0)C-             BEGIN
  56            0)—
```

```
57          0)—              {read in the numbers, one by one}
58          0)—
59        8 0)—              FOR i := 1 TO array_size DO
60          0)D-               BEGIN
61        9 0)—                  write (output,'Input number ',i:3,': ');
62       10 0)—                  force (output);
63       11 0)—                  readln (input,number_array[i])
64          0)-D               END; {FOR}
65          0)—
66          0)—              {now sort the numbers - use a bubble sort}
67          0)—
68       12 0)—              j  := array_size - 1;              {init ending index}
69          0)—
70          0)D-             REPEAT
71       13 0)—                 exchange  := false;            {show no xchng yet this pass}
72          0)—
73       14 0)—                 FOR i := 1 TO j DO
74       15 0)—                    IF number_array[i] > number_array[i+1] THEN
75          0)E-                      BEGIN                   {switch elements i and j}
76       16 0)—                          temp                := number_array[i];
77       17 0)—                          number_array[i]     := number_array[i+1];
78       18 0)—                          number_array[i+1]   := temp;
79          0)—
80       19 0)—                          exchange            := true;    {show made an xchng}
81          0)-E                     END; {THEN and FOR}
82          0)—
83       20 0)—                 j  := j - 1                    {change last index}
84       21 0)-D             UNTIL  (NOT exchange) OR (j < 1);
85          0)—
86          0)—              {now output the results}
87          0)—
88       22 0)—              writeln (output);  writeln (output);
89       24 0)—              writeln (output,'Numbers in sorted order are:');
90          0)—
91       25 0)—              FOR i := 1 TO array_size DO
92       26 0)—                 writeln (output,number_array[i]:5)
93          0)—
94          0)-C          END {THEN}
95       27 0)-B        UNTIL array_size <= 0;
96          0)—
97          0)—      {finish up}
98          0)—
99       28 0)—      writeln (output);   writeln (output);
100      30 0)—      writeln (output,'Done - Thank You')
101         0)-A   END. {sort}
```

**** No Error(s) and No Warning(s) detected

**** 101 Lines 1 Procedures

**** 365 Pcode instructions

8.1.2 Phase 2 Listing

```
                        *Motorola Pascal  2.00   SORT    .SA 03/26/82 15:32:23                    0
                        *{-------------------------------------------------------------}          1
                        *{                                                             }          2
                        *{                            S O R T                          }          3
                        *{                                                             }          4
                        *{        This program demonstrates the linking of a Pascal program   }   5
                        *{        with an assembly language subroutine.                }          6
                        *{                                                             }          7
                        *{        The driver program simply asks for an array of numbers,  }     8
                        *{        one by one, sorts the numbers in increasing numerical    }     9
                        *{        order, and prints the results.                       }          10
                        *{                                                             }          11
                        *{        An assembly language routine is used to force output     }     12
                        *{        to a text file without having to do a writeln.  This     }     13
                        *{        allows for prompting for input and having the input      }     14
                        *{        entered on the same line.                            }          15
                        *{                                                             }          16
                        *{-------------------------------------------------------------}          17
                        *                                                                         18
                        *PROGRAM  sort (input,output);                                            19
                        *                                                                         20
                        *   CONST                                                                 21
                        *        XREF    8:.PLJSR
                        *      max_array_size  = 5000;                  {maximum array size}      22
                        *                                                                         23
                        *   TYPE                                                                  24
                        *      index_range     = 1..max_array_size;      {range of indices into array}  25
                        *                                                                         26
                        *   VAR                                                                   27
                        *      number_array:     ARRAY [index_range] of integer;    {the array}    28
                        *      array_size:       0..max_array_size;        {actual size of array}   29
                        *      i,j:              index_range;              {indices into array}     30
                        *      temp:             integer;                  {used for swaping elements}  31
                        *      exchange:         boolean;                  {any exchanges in last pass?}  32
                        *                                                                         33
                        *   {declare the needed assembly language routine as external}            34
                        *                                                                         35
                        *   PROCEDURE  force (VAR fil: text);  FORWARD;                           36
                        *                                                                         37
                        *   {$A=2}                                                                38
                        *                                                                         39
                        *   {the main program starts here}                                        40
                        *                                                                         41
                        *   BEGIN {sort}                                                          42
                        *                                                                         43
                        *      REPEAT                                    {loop for each array}    44
00000000                .PMAIN    EQU   *
00000000    0001                  DC.W    1
00000002    9EFC 4E46             SUB     #20038,A7
00000006    7401                  MOVEQ   #1,D2
00000008    7205                  MOVEQ   #5,D1
0000000A    7000                  MOVEQ   #0,D0
0000000C    41ED FFF0             LEA     -16(A5),A0
                                  XREF    8:.PIFD
00000010    4EAB ****             JSR     .PIFD-.PLJSR(A3)
00000014    41ED FFF0             LEA     -16(A5),A0
                                  XREF    8:.PRWT
00000018    4EAB ****             JSR     .PRWT-.PLJSR(A3)
0000001C    7401                  MOVEQ   #1,D2
0000001E    7206                  MOVEQ   #6,D1
00000020    7000                  MOVEQ   #0,D0
00000022    41ED FFF8             LEA     -8(A5),A0
00000026    4EAB ****             JSR     .PIFD-.PLJSR(A3)
0000002A    41ED FFF8             LEA     -8(A5),A0
                                  XREF    8:.PRST
0000002E    4EAB ****             JSR     .PRST-.PLJSR(A3)
                        *                                                                        45
                        *         {ask for size of the array}                                    46
                        *                                                                        47
                        *         writeln (output);   writeln(output);                           48
00000032                L3        EQU     *
00000032    41ED FFF0             LEA     -16(A5),A0
                                  XREF    8:.PWLN
00000036    4EAB ****             JSR     .PWLN-.PLJSR(A3)
0000003A    41ED FFF0             LEA     -16(A5),A0
0000003E    4EAB ****             JSR     .PWLN-.PLJSR(A3)
                        *         write (output,'Input size of array (0 to quit): ');            49
00000042    486D FFF0             PEA     -16(A5)
                                  XREF    8:.PLDCS
00000046    4EAB ****             JSR     .PLDCS-.PLJSR(A3)
0000004A    0021                  DC.W    33
0000004C                          DC.B    'Input size of array (0 to quit): '
0000006E    4267                  CLR     -(A7)
```

```
                                        XREF    8:.PWRS
00000070    4EAB ****                   JSR     .PWRS-.PLJSR(A3)
                                *       force (output);                                         50
00000074    2F08                        MOVE.L  A0,-(A7)
00000076    4E93                        JSR     (A3)
00000078    ********                    DC.L    USER1-*
                                *                                                               51
                                *       readln (input,array_size);          {get size of array} 52
0000007C    43ED B1CE                   LEA     -20018(A5),A1
00000080    41ED FFF8                   LEA     -8(A5),A0
                                        XREF    8:.PRDI
00000084    4EAB ****                   JSR     .PRDI-.PLJSR(A3)
                                        XREF    8:.PRLN
00000088    4EAB ****                   JSR     .PRLN-.PLJSR(A3)
                                *                                                               53
                                *       IF array_size > 0 THEN                                  54
0000008C    4A6D B1CE                   TST     -20018(A5)
00000090    6F**                        BLE     L4
                                *          BEGIN                                                55
                                *                                                               56
                                *             {read in the numbers, one by one}                 57
                                *                                                               58
                                *          FOR i := 1 TO array_size DO                          59
00000092-0094 3B7C 0001 B1CA            MOVE    #1,-20022(A5)
00000098-009A 3B6D B1CE B1C2            MOVE    -20018(A5),-20030(A5)
0000009E-00A0 60**                      BRA     L6
000000A0-00A4                   L5      EQU     *
                                *           BEGIN                                               60
                                *              write (output,'Input number ',i:3,': ');         61
000000A0-00A4 486D FFF0                 PEA     -16(A5)
000000A4-00A8 4EAB ****                 JSR     .PLDCS-.PLJSR(A3)
000000A8-00AC 000D                      DC.W    13
000000AA-00AE                           DC.B    'Input number '
000000B8-00BC 4267                      CLR     -(A7)
000000BA-00BE 4EAB ****                 JSR     .PWRS-.PLJSR(A3)
000000BE-00C2 7203                      MOVEQ   #3,D1
000000C0-00C4 302D B1CA                 MOVE    -20022(A5),D0
                                        XREF    8:.PWRI
000000C4-00C8 4EAB ****                 JSR     .PWRI-.PLJSR(A3)
000000C8-00CC 2F08                      MOVE.L  A0,-(A7)
000000CA-00CE 2F3C 00023A20             MOVE.L  #145952,-(A7)
000000D0-00D4 4267                      CLR     -(A7)
000000D2-00D6 4EAB ****                 JSR     .PWRS-.PLJSR(A3)
                                *              force (output);                                  62
000000D6-00DA 2F08                      MOVE.L  A0,-(A7)
000000D8-00DC 4E93                      JSR     (A3)
000000DA-00DE ********                  DC.L    USER1-*
                                *              readln (input,number_array[i])                   63
                                *           END; {FOR}                                          64
000000DE-00E2 302D B1CA                 MOVE    -20022(A5),D0
000000E2-00E6 E540                      ASL     #2,D0
000000E4-00E8 41ED B1CC                 LEA     -20020(A5),A0
000000E8-00EC 43F0 0000                 LEA     0(A0,D0),A1
000000EC-00F0 41ED FFF8                 LEA     -8(A5),A0
                                        XREF    8:.PRDJ
000000F0-00F4 4EAB ****                 JSR     .PRDJ-.PLJSR(A3)
000000F4-00F8 4EAB ****                 JSR     .PRLN-.PLJSR(A3)
000000F8-00FC 526D B1CA                 ADDQ    #1,-20022(A5)
000000FC-0100 69**                      BVS     L7
000000FE-0104                   L6      EQU     *
000000FE-0104 302D B1C2                 MOVE    -20030(A5),D0
00000102-0108 B06D B1CA                 CMP     -20022(A5),D0
00000106-010C 6C**                      BGE     L5
00000108-0110                   L7      EQU     *
                                *                                                               65
                                *       {now sort the numbers - use a bubble sort}              66
                                *                                                               67
                                *       j  := array_size - 1;           {init ending index}     68
00000108-0110 302D B1CE                 MOVE    -20018(A5),D0
0000010C-0114 5340                      SUBQ    #1,D0
0000010E-0116 3B40 B1CC                 MOVE    D0,-20020(A5)
                                *                                                               69
                                *       REPEAT                                                  70
                                *          exchange  := false;          {show no xchng yet this pass} 71
00000112-011A                   L8      EQU     *
00000112-011A 422D B1C5                 CLR.B   -20027(A5)
                                *                                                               72
                                *          FOR i := 1 TO j DO                                   73
00000116-011E 3B7C 0001 B1CA            MOVE    #1,-20022(A5)
0000011C-0124 3B6D B1CC B1C2            MOVE    -20020(A5),-20030(A5)
00000122-012A 60**                      BRA     L10
00000124-012E                   L9      EQU     *
                                *              IF number_array[i] > number_array[i+1] THEN      74
00000124-012E 302D B1CA                 MOVE    -20022(A5),D0
00000128-0132 E540                      ASL     #2,D0
0000012A-0134 41ED B1CC                 LEA     -20020(A5),A0
0000012E-0138 322D B1CA                 MOVE    -20022(A5),D1
00000132-013C E541                      ASL     #2,D1
00000134-013E 43ED B1D0                 LEA     -20016(A5),A1
```

```
00000138-0142 2431 1000                     MOVE.L  0(A1,D1),D2
0000013C-0146 B4B0 0000                     CMP.L   0(A0,D0),D2
00000140-014A 6C**                          BGE     L12
                                      *                       BEGIN                       {switch elements i and j}          75
                                      *                         temp                := number_array[i];                      76
00000142-014E 2B70 0000 B1C6                MOVE.L  0(A0,D0),-20026(A5)
                                      *                         number_array[i]     := number_array[i+1];                    77
00000148-0154 21B1 1000 0000                MOVE.L  0(A1,D1),0(A0,D0)
                                      *                         number_array[i+1]   := temp;                                 78
0000014E-015A 23AD B1C6 1000                MOVE.L  -20026(A5),0(A1,D1)
                                      *                                                                                      79
                                      *                         exchange            := true;    {show made an xchng}         80
00000154-0160 1B7C 0001 B1C5                MOVE.B  #1,-20027(A5)
                                      *                       END; {THEN and FOR}                                            81
0000015A-0166                         L12   EQU     *
0000015A-0166 526D B1CA                     ADDQ    #1,-20022(A5)
0000015E-016A 69**                          BVS     L11
00000160-016E                         L10   EQU     *
00000160-016E 302D B1C2                     MOVE    -20030(A5),D0
00000164-0172 B06D B1CA                     CMP     -20022(A5),D0
00000168-0176 6C**                          BGE     L9
0000016A-017A                         L11   EQU     *
                                      *                                                                                      82
                                      *                   j  := j - 1                     {change last index}                83
                                      *                 UNTIL  (NOT exchange) OR (j < 1);                                    84
0000016A-017A 536D B1CC                     SUBQ    #1,-20020(A5)
0000016E-017E 4A2D B1C5                     TST.B   -20027(A5)
00000172-0182 67**                          BEQ     L13
00000174-0186 0C6D 0001 B1CC                CMP     #1,-20020(A5)
0000017A-018C 6C**                          BGE     L8
0000017C-0190                         L13   EQU     *
                                      *                                                                                      85
                                      *                 {now output the results}                                             86
                                      *                                                                                      87
                                      *                 writeln (output);  writeln (output);                                 88
0000017C-0190 41ED FFF0                     LEA     -16(A5),A0
00000180-0194 4EAB ****                     JSR     .PWLN-.PLJSR(A3)
00000184-0198 41ED FFF0                     LEA     -16(A5),A0
00000188-019C 4EAB ****                     JSR     .PWLN-.PLJSR(A3)
                                      *                 writeln (output,'Numbers in sorted order are:');                     89
0000018C-01A0 486D FFF0                     PEA     -16(A5)
00000190-01A4 4EAB ****                     JSR     .PLDCS-.PLJSR(A3)
00000194-01A8 001C                          DC.W    28
00000196-01AA                               DC.B    'Numbers in sorted order are:'
000001B2-01C6 4267                          CLR     -(A7)
000001B4-01C8 4EAB ****                     JSR     .PWRS-.PLJSR(A3)
000001B8-01CC 4EAB ****                     JSR     .PWLN-.PLJSR(A3)
                                      *                                                                                      90
                                      *                 FOR i := 1 TO array_size DO                                          91
000001BC-01D0 3B7C 0001 B1CA                MOVE    #1,-20022(A5)
000001C2-01D6 3B6D B1CE B1C2                MOVE    -20018(A5),-20030(A5)
000001C8-01DC 60**                          BRA     L15
000001CA-01E0                         L14   EQU     *
                                      *                   writeln (output,number_array[i]:5)                                 92
                                      *                                                                                      93
                                      *               END {THEN}                                                             94
000001CA-01E0 302D B1CA                     MOVE    -20022(A5),D0
000001CE-01E4 E540                          ASL     #2,D0
000001D0-01E6 41ED B1CC                     LEA     -20020(A5),A0
000001D4-01EA 7205                          MOVEQ   #5,D1
000001D6-01EC 2030 0000                     MOVE.L  0(A0,D0),D0
000001DA-01F0 41ED FFF0                     LEA     -16(A5),A0
                                            XREF    8:.PWRJ
000001DE-01F4 4EAB ****                     JSR     .PWRJ-.PLJSR(A3)
000001E2-01F8 4EAB ****                     JSR     .PWLN-.PLJSR(A3)
000001E6-01FC 526D B1CA                     ADDQ    #1,-20022(A5)
000001EA-0200 69**                          BVS     L16
000001EC-0204                         L15   EQU     *
000001EC-0204 302D B1C2                     MOVE    -20030(A5),D0
000001F0-0208 B06D B1CA                     CMP     -20022(A5),D0
000001F4-020C 6C**                          BGE     L14
000001F6-0210                         L16   EQU     *
                                      *               UNTIL array_size <= 0;                                                 95
000001F6-0210                         L4    EQU     *
000001F6-0210 4A6D B1CE                     TST     -20018(A5)
000001FA-0214 6E00 ****                     LBGT    L3
                                      *                                                                                      96
                                      *           {finish up}                                                                97
                                      *                                                                                      98
                                      *           writeln (output);   writeln (output);                                      99
000001FE-0218 41ED FFF0                       LEA     -16(A5),A0
00000202-021C 4EAB ****                       JSR     .PWLN-.PLJSR(A3)
00000206-0220 41ED FFF0                       LEA     -16(A5),A0
0000020A-0224 4EAB ****                       JSR     .PWLN-.PLJSR(A3)
                                      *           writeln (output,'Done - Thank You')                                        100
                                      *       END. {sort}                                                                    101
0000020E-0228 486D FFF0                       PEA     -16(A5)
00000212-022C 4EAB ****                       JSR     .PLDCS-.PLJSR(A3)
```

```
00000216-0230 0010                      DC.W    16
00000218-0232                           DC.B    'Done - Thank You'
00000228-0242 4267                      CLR     -(A7)
0000022A-0244 4EAB ****                 JSR     .PWRS-.PLJSR(A3)
0000022E-0248 4EAB ****                 JSR     .PWLN-.PLJSR(A3)
00000232-024C 41ED FFF0                 LEA     -16(A5),A0
                                        XREF    8:.PCLO
00000236-0250 4EAB ****                 JSR     .PCLO-.PLJSR(A3)
0000023A-0254 41ED FFF8                 LEA     -8(A5),A0
0000023E-0258 4EAB ****                 JSR     .PCLO-.PLJSR(A3)
00000242-025C 4267                      CLR     -(A7)
00000244-025E 4E4E                      TRAP    #14
00000246-0260 0004                      DC.W    4
                                        END

                        *** Total of 586 bytes generated
```

8.1.3 Assembly Listing

```
MOTOROLA M68000 ASM              VOL1:  21.        .FORCE    .SA

  2                         FORCE    IDNT     1,1                  FORCE OUTPUT PROCEDURE FOR PASCAL
  3                         *
  4                         *                 F O R C E
  5                         *
  6                         *       PROCEDURE  FORCE (FIL: TEXT)
  7                         *
  8                         *       THIS ASSEMBLY LANGUAGE ROUTINE INTERFACES TO A PASCAL PROGRAM
  9                         *       AND FORCES WHATEVER IS IN THE BUFFER FOR THE FILE POINTED TO BY
 10                         *       FIL OUT TO THE FILE WITHOUT A CARRIAGE RETURN.
 11                         *
 12                         *       THOSE REGISTERS USED ARE A0, A1, A2, A4, D1, D2, AND D4.
 13                         *
 14                         *
 15                         *       XDEF THE NAME OF THIS ROUTINE
 16                         *
 17                           XDEF     FORCE
 18                         *
 19                         *
 20                         *       THE FOLLOWING ARE OFFSETS INTO A PASCAL PARAMETER BLOCK
 21                         *
 22           00000008      FUNCTION EQU      8                    IOS FUNCTION CODE
 23           00000014      BUFSTART EQU      20                   ADDRESS OF START OF BUFFER
 24           00000018      BUFEND   EQU      24                   ADDRESS OF END OF BUFFER
 25                         *
 26                         *       THE FOLLOWING ARE THE NECESSARY EQUATES FOR IOS
 27                         *
 28           00000002      IOS      EQU      2                    THE TRAP NUMBER FOR IOS
 29                         *
 30           00020008      FORCEFUNC EQU     $00020008            WRITE, NO FORMAT, WAIT, NEXT RECORD
 31                         *
 32                         *
 33                         *       THE CODE STARTS HERE
 34                         *
 35           00000009                SECTION  9                    SET UP THE SECTION
 36                         *
 37 9         00000000      FORCE    EQU      *
 38                         *
 39 9 00000000 285F                   MOVE.L   (A7)+,A4             SAVE THE RETURN ADDRESS
 40 9 00000002 245F                   MOVE.L   (A7)+,A2             GET ADDRESS OF FILE POINTER
 41 9 00000004 2212                   MOVE.L   (A2),D1              GET COMPONENT POINTER
 42 9 00000006 226A0004               MOVE.L   4(A2),A1             GET ADDRESS OF PARAMETER BLOCK
 43                         *
 44 9 0000000A 24290008               MOVE.L   FUNCTION(A1),D2      SAVE OLD FUNCTION/OPTIONS
 45 9 0000000E 237C00020008           MOVE.L   #FORCEFUNC,FUNCTION(A1) SET UP NEW FUNCTION/OPTIONS
              0008
 46                         *
 47 9 00000016 26290018               MOVE.L   BUFEND(A1),D3        SAVE OLD END ADDRESS
 48                         *
 49 9 0000001A 5381                   SUB.L    #1,D1                SET UP...
```

```
  50 9 0000001C 23410018                MOVE.L   D1,BUFEND(A1)          END ADDRESS.
  51                          *
  52 9 00000020 41E90008                LEA      FUNCTION(A1),A0        CALL THE...
  53 9 00000024 4E42                    TRAP     #IOS                   IOS.
  54                          *
  55 9 00000026 23430018                MOVE.L   D3,BUFEND(A1)          RESTORE THE END ADDRESS.
  56 9 0000002A 23420008                MOVE.L   D2,FUNCTION(A1)        RESTORE OLD FUNCTION/OPTION
  57 9 0000002E 24A90014                MOVE.L   BUFSTART(A1),(A2)      RESET COMPONENT POINTER
  58                          *
  59 9 00000032 4ED4                    JMP      (A4)                   RETURN
  60                          *
  61                                    END

****** TOTAL ERRORS      0--
****** TOTAL WARNINGS    0--


SYMBOL TABLE LISTING

SYMBOL NAME     SECT   VALUE       SYMBOL NAME     SECT   VALUE


BUFEND                 00000018     FORCEFUN               00020008
BUFSTART               00000014     FUNCTION               00000008
FORCE      XDEF   9    00000000     IOS                    00000002
```

8.1.4 Linkage Editor Listing

```
Motorola M68000 Linkage Editor


Command Line:

LINK SORT/FORCE,PASPROG.LO,PASPROG.LL;HIMUXL=PASCALIB

Options in Effect:  -A,-B,-D,H,I,L,M,O,P,-Q,-R,-S,U,-W,X

User Commands: None

Object Module Header Information:

Module      Ver Rev Language Date     Time     Creation File Name

SORT          1   0 Pascal   03/26/82 15:36:30 FIX:21..SORT.PO
            M68000 Pascal object from  2.00 resident compiler

FORCE         1   1 Assembly 02/16/82 13:50:57 VOL1:21..FORCE.SA
            FORCE OUTPUT PROCEDURE FOR PASCAL

INIT          2  11 Assembly 02/08/82 14:34:09 FIX:103.UTILRR.INIT.SA
            68K PASCAL INITIALIZE RUNTIME ENVIRONMENT

SYMFLAG       0   0 Assembly 01/19/82 17:10:19 FIX:103.UTILRR.SYMBUG.SA
            SYMBUG FLAG

TRAPS         2  11 Assembly 01/19/82 17:39:43 FIX:103.UTILRR.TRAPS.SA
            68K PASCAL TRAP HANDLING RUNTIME SUBROUTINES

OPTION        2   0 Assembly 01/19/82 17:26:41 FIX:103.UTILRR.OPTION.SA
            68K PASCAL OPTION PROCESSOR SUBROUTINE
```

```
CLSCOD      2   0 Assembly 01/18/82 17:54:13 FIX:103.IOSRR.CLSCOD.SA
           68K PASCAL CHARACTER CLASS CODE TABLE

ALSTS       2   0 Assembly 01/21/82 10:25:59 FIX:103.UTILRR.ALSTS.SA
           68K PASCAL ALLOCATE STACK SUBROUTINE

CLO         2   0 Assembly 01/19/82 17:48:31 FIX:103.IORR.CLO.SA
           68K PASCAL CLOSE FILE SUBROUTINE

IFD         2   0 Assembly 01/19/82 17:52:41 FIX:103.IORR.IFD.SA
           68K PASCAL INITIALIZE FILE DESCRIPTOR SUBROUTINE

RST         2   0 Assembly 01/19/82 17:56:59 FIX:103.IORR.RST.SA
           68K PASCAL RESET FILE SUBROUTINE

RWT         2   0 Assembly 01/19/82 17:58:12 FIX:103.IORR.RWT.SA
           68K PASCAL REWRTE FILE SUBROUTINE

ACCPER      2   0 Assembly 01/19/82 17:41:51 FIX:103.IORR.ACCPER.SA
           68K PASCAL ACCESS PERMISSION SET UP SUBROUTINE

CALCLU      2   0 Assembly 01/19/82 17:46:37 FIX:103.IORR.CALCLU.SA
           68K PASCAL CALCULATE LOGICAL UNIT SUBROUTINE

EDTFIL      2   0 Assembly 01/18/82 17:54:56 FIX:103.IOSRR.EDTFIL.SA
           68K PASCAL EDIT FILE NAME SUBROUTINE

PRGBUF      2   0 Assembly 01/19/82 17:54:34 FIX:103.IORR.PRGBUF.SA
           68K PASCAL PURGE BUFFER SUBROUTINE

STDFLT      2   0 Assembly 01/19/82 17:59:15 FIX:103.IORR.STDFLT.SA
           68K PASCAL SET START FILE DEFAULTS SUBROUTINE

DFLT        2   0 Assembly 01/19/82 17:50:37 FIX:103.IORR.DFLT.SA
           68K PASCAL SET FILE DEFAULTS SUBROUTINE

WLN         2   0 Assembly 01/18/82 18:14:12 FIX:103.IOSRR.WLN.SA
           68K PASCAL WRITELN SUBROUTINE

WRI         2   0 Assembly 01/18/82 18:16:21 FIX:103.IOSRR.WRI.SA
           68K PASCAL WRITE INTEGER SUBROUTINES

WRSWRV      2   0 Assembly 01/18/82 18:17:17 FIX:103.IOSRR.WRSWRV.SA
           68K PASCAL WRITE STRING AND VECTOR SUBROUTINES

WRTBUF      2   0 Assembly 01/19/82 18:00:15 FIX:103.IORR.WRTBUF.SA
           68K PASCAL WRITE BUFFER SUBROUTINE

LBLKS       2   0 Assembly 01/18/82 18:02:09 FIX:103.IOSRR.LBLKS.SA
           68K PASCAL WRITE LEADING BLANKS SUBROUTINE

IWPTR       2   0 Assembly 01/18/82 18:01:25 FIX:103.IOSRR.IWPTR.SA
           68K PASCAL INCREMENT TEXT FILE COMPONENT PTR SUBROUTINE

RLN         2   0 Assembly 01/18/82 18:12:46 FIX:103.IOSRR.RLN.SA
           68K PASCAL READLN SUBROUTINE

RDI         2   0 Assembly 01/18/82 18:08:12 FIX:103.IOSRR.RDI.SA
           68K PASCAL READ TWO BYTE INTEGER SUBROUTINE

RDJ         2   0 Assembly 01/18/82 18:12:02 FIX:103.IOSRR.RDJ.SA
           68K PASCAL READ FOUR BYTE INTEGER SUBROUTINE

RDINT       2   0 Assembly 01/18/82 18:08:55 FIX:103.IOSRR.RDINT.SA
           68K PASCAL READ UNSIZED INTEGER SUBROUTINE

SBLKS       2   0 Assembly 01/18/82 18:13:30 FIX:103.IOSRR.SBLKS.SA
           68K PASCAL SKIP LEADING BLANKS SUBROUTINE

GETCH       2   0 Assembly 01/18/82 17:58:41 FIX:103.IOSRR.GETCH.SA
           68K PASCAL GET CHARACTER FROM INPUT BUFFER SUBROUTINE

IRPTR       2   0 Assembly 01/18/82 18:00:43 FIX:103.IOSRR.IRPTR.SA
           68K PASCAL INCREMENT INPUT COMPONENT PTR SUBROUTINE
```

```
GETINT      2   0 Assembly 01/18/82 17:59:23 FIX:103.IOSRR.GETINT.SA
           68K PASCAL COLLECT INTEGER DIGITS SUBROUTINE

MAKINT      2   0 Assembly 01/18/82 18:02:50 FIX:103.IOSRR.MAKINT.SA
           68K PASCAL MAKE AN INTEGER FROM DIGITS SUBROUTINE

RDBUF       2   0 Assembly 01/25/82 10:29:34 FIX:103.IORR.RDBUF.SA
           68K PASCAL READ BUFFER FROM FILE SUBROUTINE

ASGNF       2   0 Assembly 01/19/82 17:44:45 FIX:103.IORR.ASGNF.SA
           68K PASCAL ASSIGN FILE SUBROUTINE

BUFSZ       2   0 Assembly 01/19/82 17:45:43 FIX:103.IORR.BUFSZ.SA
           68K PASCAL FIGURE OUT BUFFER SIZE SUBROUTINE

CLOSE       2   0 Assembly 01/19/82 17:49:31 FIX:103.IORR.CLOSE.SA
           68K PASCAL FILE CLOSE SUBROUTINES

CFLDAD      2   0 Assembly 01/19/82 17:47:36 FIX:103.IORR.CFLDAD.SA
           68K PASCAL CALCLUATE FIELD ADDRESS SUBROUTINE

FLSCN       2   0 Assembly 01/19/82 17:51:39 FIX:103.IORR.FLSCN.SA
           68K PASCAL FILE LIST SCAN SUBROUTINE

LDC         2   0 Assembly 01/20/82 12:35:13 FIX:103.UTILRR.LDC.SA
           68K PASCAL LOAD CONSTANT STRING AND VECTOR SUBROUTINE


Load Map:


Segment SEG1(R): 00000000 000017FF 8,9,10,11,12,13,14
Module      S   T  Start     End       Externally Defined Symbols

INIT        8      00000000  0000036B  .PLJSR     00000364  .PINIT     00000000
SYMFLAG     8      0000036C  0000036D  .PSYMBUG   0000036C
TRAPS       8      0000036E  000004FF  .PADDRER   000004D6  .PVBUSER   000004B8
                                       .PVCHKI    0000049A  .PVTRAPD   0000049A
                                       .PVTRAPE   0000036E  .PVTRAPV   00000480
                                       .PVZDIV    00000466
OPTION      8      00000500  00000659  .POPTION   00000500
CLSCOD      8      0000065A  000006D9  .PCLSCOD   0000065A
ALSTS       8      000006DA  00000705  .PALSTS    000006DA  .PALSTSL   000006DC
CLO         8      00000706  00000721  .PCLO      00000706
IFD         8      00000722  00000937  .PIFD      00000722
RST         8      00000938  000009E9  .PRST      00000938
RWT         8      000009EA  00000A69  .PRWT      000009EA
ACCPER      8      00000A6A  00000A8B  .PACCPER   00000A6A
CALCLU      8      00000A8C  00000ADD  .PCALCLU   00000A8C
EDTFIL      8      00000ADE  00000EBF  .PEDTFIL   00000ADE
PRGBUF      8      00000EC0  00000ED9  .PPRGBUF   00000EC0
STDFLT      8      00000EDA  00000F37  .PSTDFLT   00000EDA
DFLT        8      00000F38  00000F93  .PDFLT     00000F38
WLN         8      00000F94  00000FA3  .PWLN      00000F94
WRI         8      00000FA4  00001031  .PWRI      00000FA6  .PWRH      00000FA4
                                       .PWRJ      00000FA8
WRSWRV      8      00001032  0000107B  .PWRS      0000103C  .PWRV      00001032
WRTBUF      8      0000107C  000010DF  .PWRTBUF   0000107C
LBLKS       8      000010E0  000010F1  .PLBLKS    000010E0
IWPTR       8      000010F2  00001105  .PIWPTR    000010F2
RLN         8      00001106  00001141  .PRLN      00001106
RDI         8      00001142  0000114B  .PRDI      00001142
RDJ         8      0000114C  00001155  .PRDJ      0000114C
RDINT       8      00001156  000011B7  .PRDINT    00001156
SBLKS       8      000011B8  000011C9  .PSBLKS    000011B8
GETCH       8      000011CA  000011E7  .PGETCH    000011CA
IRPTR       8      000011E8  00001205  .PIRPTR    000011E8
GETINT      8      00001206  00001319  .PGETINT   00001206
MAKINT      8      0000131A  00001377  .PMAKINT   0000131A
RDBUF       8      00001378  0000146F  .PRDBUF    00001378
ASGNF       8      00001470  00001493  .PASGNF    00001470
BUFSZ       8      00001494  000014A3  .PBUFSZ    00001494
CLOSE       8      000014A4  000014D5  .PCLOSE    000014A4  .PCLOSPL   000014C4
```

```
CFLDAD      8      000014D6  000014FD  .PCFLDAD    000014D6
FLSCN       8      000014FE  00001527  .PFLSCN     000014FE
LDC         8      00001528  00001551  .PLDCS      00001528  .PLDCV      0000152C
FINIT       8  C   00001552  00001553
SORT        9      00001554  0000179D  .PMAIN      00001554
FORCE       9      0000179E  000017D1  FORCE       0000179E


Segment SEG2: 00001800 000079FF 15
Module      S   T  Start     End       Externally Defined Symbols

SORT        15     00001800  000079FF  .PZMAIN     000079FE

Table of Externally Defined Symbols:

Name        Address   Module      Displ     Sect Seg   Library      Input

.PACCPER    00000A6A  ACCPER      00000000   8  SEG1  PASCALIB.RO
.PADDRER    000004D6  TRAPS       00000168   8  SEG1  PASCALIB.RO
.PALSTS     000006DA  ALSTS       00000000   8  SEG1  PASCALIB.RO
.PALSTSL    000006DC  ALSTS       00000002   8  SEG1  PASCALIB.RO
.PASGNF     00001470  ASGNF       00000000   8  SEG1  PASCALIB.RO
.PBUFSZ     00001494  BUFSZ       00000000   8  SEG1  PASCALIB.RO
.PCALCLU    00000A8C  CALCLU      00000000   8  SEG1  PASCALIB.RO
.PCFLDAD    000014D6  CFLDAD      00000000   8  SEG1  PASCALIB.RO
.PCLO       00000706  CLO         00000000   8  SEG1  PASCALIB.RO
.PCLOSE     000014A4  CLOSE       00000000   8  SEG1  PASCALIB.RO
.PCLOSPL    000014C4  CLOSE       00000020   8  SEG1  PASCALIB.RO
.PCLSCOD    0000065A  CLSCOD      00000000   8  SEG1  PASCALIB.RO
.PDFLT      00000F38  DFLT        00000000   8  SEG1  PASCALIB.RO
.PEDTFIL    00000ADE  EDTFIL      00000000   8  SEG1  PASCALIB.RO
.PFLSCN     000014FE  FLSCN       00000000   8  SEG1  PASCALIB.RO
.PGETCH     000011CA  GETCH       00000000   8  SEG1  PASCALIB.RO
.PGETINT    00001206  GETINT      00000000   8  SEG1  PASCALIB.RO
.PIFD       00000722  IFD         00000000   8  SEG1  PASCALIB.RO
.PINIT      00000000  INIT        00000000   8  SEG1  PASCALIB.RO
.PIRPTR     000011E8  IRPTR       00000000   8  SEG1  PASCALIB.RO
.PIWPTR     000010F2  IWPTR       00000000   8  SEG1  PASCALIB.RO
.PLBLKS     000010E0  LBLKS       00000000   8  SEG1  PASCALIB.RO
.PLDCS      00001528  LDC         00000000   8  SEG1  PASCALIB.RO
.PLDCV      0000152C  LDC         00000004   8  SEG1  PASCALIB.RO
.PLJSR      00000364  INIT        00000364   8  SEG1  PASCALIB.RO
.PMAIN      00001554  SORT        00000000   9  SEG1                SORT     .RO
.PMAKINT    0000131A  MAKINT      00000000   8  SEG1  PASCALIB.RO
.POPTION    00000500  OPTION      00000000   8  SEG1  PASCALIB.RO
.PPRGBUF    00000EC0  PRGBUF      00000000   8  SEG1  PASCALIB.RO
.PRDBUF     00001378  RDBUF       00000000   8  SEG1  PASCALIB.RO
.PRDI       00001142  RDI         00000000   8  SEG1  PASCALIB.RO
.PRDINT     00001156  RDINT       00000000   8  SEG1  PASCALIB.RO
.PRDJ       0000114C  RDJ         00000000   8  SEG1  PASCALIB.RO
.PRLN       00001106  RLN         00000000   8  SEG1  PASCALIB.RO
.PRST       00000938  RST         00000000   8  SEG1  PASCALIB.RO
.PRWT       000009EA  RWT         00000000   8  SEG1  PASCALIB.RO
.PSBLKS     000011B8  SBLKS       00000000   8  SEG1  PASCALIB.RO
.PSTDFLT    00000EDA  STDFLT      00000000   8  SEG1  PASCALIB.RO
.PSYMBUG    0000036C  SYMFLAG     00000000   8  SEG1  PASCALIB.RO
.PVBUSER    000004B8  TRAPS       0000014A   8  SEG1  PASCALIB.RO
.PVCHKI     0000049A  TRAPS       0000012C   8  SEG1  PASCALIB.RO
.PVTRAPD    0000049A  TRAPS       0000012C   8  SEG1  PASCALIB.RO
0PVTRAPE    0000036E  TRAPS       00000000   8  SEG1  PASCALIB.RO
.PVTRAPV    00000480  TRAPS       00000112   8  SEG1  PASCALIB.RO
.PVZDIV     00000466  TRAPS       000000F8   8  SEG1  PASCALIB.RO
.PWLN       00000F94  WLN         00000000   8  SEG1  PASCALIB.RO
.PWRH       00000FA4  WRI         00000000   8  SEG1  PASCALIB.RO
.PWRI       00000FA6  WRI         00000002   8  SEG1  PASCALIB.RO
.PWRJ       00000FA8  WRI         00000004   8  SEG1  PASCALIB.RO
.PWRS       0000103C  WRSWRV      0000000A   8  SEG1  PASCALIB.RO
.PWRTBUF    0000107C  WRTBUF      00000000   8  SEG1  PASCALIB.RO
.PWRV       00001032  WRSWRV      00000000   8  SEG1  PASCALIB.RO
.PZMAIN     000079FE  SORT        000061FE  15  SEG2                SORT     .RO
FORCE       0000179E  FORCE       00000000   9  SEG1                FORCE    .RO
```

```
Unresolved References: None

Multiply Defined Symbols: None

Lengths (in bytes):

     Segment      Hex          Decimal

      SEG1     00001800          6144
      SEG2     00006200         25088
Total Length   00007A00         31232

   No Errors
   No Warnings

Load module has been created.
```

8.2 PROGRAM FOR VERSAmodule 01 EXECUTION UNDER VERSAbug

8.2.1 Phase 1 Listing

```
Line      Loc Lev BE Motorola Pascal          EXAMP2  .SA

   1(       -8)  0)-- Program Example(Output);
   2(       -8)  0)--
   3(       -8)  0)-- {*-------------------------------------------------------*
   4(       -8)  0)--  *                                                       *
   5(       -8)  0)--  *       This is a sample program designed to show        *
   6(       -8)  0)--  *       people how to load and run a Pascal program      *
   7(       -8)  0)--  *                                                       *
   8(       -8)  0)--  *-------------------------------------------------------*}
   9(       -8)  0)--
  10         1   0)A-          Begin
  11         2   0)--                    writeln(Output,'I am a program.');
  12             0)-A          End.

      **** No Error(s) and No Warning(s) detected

      **** 12 Lines 0 Procedures

      **** 48 Pcode instructions
```

8.2.2 Phase 2 Listing

```
M68000 Pascal Compiler Phase 2          EXAMP2  .PC

                                *Motorola Pascal          EXAMP2  .SA
                                *Program Example(Output);
                                *
                                *{*-------------------------------------------------------*
                                * *                                                       *
                                * *       This is a sample program designed to show       *
                                * *       people how to load and run a Pascal program     *
                                * *                                                       *
                                * *-------------------------------------------------------*}
                                *
                                *         Begin
                                          XREF    8:.PLJSR
                                *                   writeln(Output,'I am a program.');
00000000                        .PMAIN    EQU   *
00000000        0001                      DC.W    1
00000002        9EFC 0008                 SUB     #8,A7
00000006        7401                      MOVEQ   #1,D2
00000008        7205                      MOVEQ   #5,D1
0000000A        7000                      MOVEQ   #0,D0
0000000C        41ED FFF8                 LEA     -8(A5),A0
                                          XREF    8:.PIFD
00000010        4EAB ****                 JSR     .PIFD-.PLJSR(A3)
00000014        41ED FFF8                 LEA     -8(A5),A0
                                          XREF    8:.PRWT
00000018        4EAB ****                 JSR     .PRWT-.PLJSR(A3)
0000001C        486D FFF8                 PEA     -8(A5)
                                          XREF    8:.PLDCS
00000020        4EAB ****                 JSR     .PLDCS-.PLJSR(A3)
00000024        000F                      DC.W    15
00000026                                  DC.B    'I am a program.'
00000036        4267                      CLR     -(A7)
                                          XREF    8:.PWRS
00000038        4EAB ****                 JSR     .PWRS-.PLJSR(A3)
                                          XREF    8:.PWLN
0000003C        4EAB ****                 JSR     .PWLN-.PLJSR(A3)
```

```
                                    *           End.
00000040      41ED FFF8                         LEA     -8(A5),A0
                                                XREF    8:.PCLO
00000044      4EAB ****                         JSR     .PCLO-.PLJSR(A3)
00000048      4267                              CLR     -(A7)
0000004A      4E4E                              TRAP    #14
0000004C      0004                              DC.W    4
                                                END
                                    *** Total of 78 bytes generated
```

8.2.3 Linkage Editor Listing

```
Motorola M68000 Linkage Editor


Command Line:

LINK ,,EXAMP2.LL;HIMSQUX


Options in Effect:  A,-B,-D,H,I,-L,M,-O,P,Q,-R,S,U,-W,X


User Commands:

SEG SEG1(R):8-14 $1000
SEG SEG2:0,15
IN EXAMP2
LIB VM01PLIB
LIB PASCALIB
END


Object Module Header Information:

Module      Ver Rev Language Date     Time     Creation File Name

EXAMPLE       1   0 Pascal   03/26/82 16:07:07 FIX:21..EXAMP2.PC
            M68000 Pascal object from  2.00 resident compiler

INIT          0   0 Assembly 02/22/82 16:21:17 FIX:103.MOD.INIT.SA
            INIT VERSAMODULE PASCAL

TABLE         0   0 Assembly 02/22/82 16:17:53 FIX:103.MOD.TABLE.SA
            IO DEFINITIONS TABLES

START         0   0 Assembly 02/22/82 16:23:16 FIX:103.MOD.START.SA
            STARTUP ROUTINE FOR THE PASCAL PROGRAM

TRAP1         0   0 Assembly 02/22/82 16:22:07 FIX:103.MOD.TRAP1.SA
            TRAP 1 HANDLER

IO            0   0 Assembly 02/22/82 16:19:01 FIX:103.MOD.IO.SA
            IO SERVICE ROUTINES FOR VERSAMODULE

PASIO         0   0 Assembly 02/22/82 16:15:37 FIX:103.MOD.PASIO.SA
            PASCAL I/O SERVICE FOR VERSAMODULE

ILL           0   0 Assembly 02/22/82 16:22:43 FIX:103.MOD.ILL.SA
            ILLEGAL I/O HANDLER

PRINT         0   0 Assembly 02/22/82 16:24:39 FIX:103.MOD.PRINT.SA
            PRINT A CHARCTER ON THE LINE PRINTER

TRAPS         2  11 Assembly 01/19/82 17:39:43 FIX:103.UTILRR.TRAPS.SA
            68K PASCAL TRAP HANDLING RUNTIME SUBROUTINES
```

```
ALSTS       2   0 Assembly 01/21/82 10:25:59 FIX:103.UTILRR.ALSTS.SA
           68K PASCAL ALLOCATE STACK SUBROUTINE

WLN         2   0 Assembly 01/18/82 18:14:12 FIX:103.IOSRR.WLN.SA
           68K PASCAL WRITELN SUBROUTINE

WRSWRV      2   0 Assembly 01/18/82 18:17:17 FIX:103.IOSRR.WRSWRV.SA
           68K PASCAL WRITE STRING AND VECTOR SUBROUTINES

LBLKS       2   0 Assembly 01/18/82 18:02:09 FIX:103.IOSRR.LBLKS.SA
           68K PASCAL WRITE LEADING BLANKS SUBROUTINE

IWPTR       2   0 Assembly 01/18/82 18:01:25 FIX:103.IOSRR.IWPTR.SA
           68K PASCAL INCREMENT TEXT FILE COMPONENT PTR SUBROUTINE

LDC         2   0 Assembly 01/20/82 12:35:13 FIX:103.UTILRR.LDC.SA
           68K PASCAL LOAD CONSTANT STRING AND VECTOR SUBROUTINE


Load Map:


Segment SEG1(R): 00001000 000019FF 8,9,10,11,12,13,14
Module      S   T  Start     End       Externally Defined Symbols

INIT        8      00001000  00001093  .PLJSR     0000108C  .PINI      00001000
TABLE       8      00001094  0000115D  DDTLINK    00001094  CWAITCH    000010B0
                                       CWAITB     000010A8  CWAITAD    000010AC
                                       CBREAKHB   000010A4  CBREAKB    0000109C
                                       CBREAKAD   000010A0  DEBUG      000010C2
                                       OTH$DV     000010BE  .PTABLE    00001094
                                       .PVTRAP1   00001098  MEMBEG     000010C6
                                       MEMEND     000010CA  EOF        000010B2
                                       START      000010B6  STD$DV     000010BA
START       8      0000115E  0000116D  .PSTART    0000115E
TRAP1       8      0000116E  00001191  TRP1HNDL   0000116E
IO          8      00001192  000013F1  AWRT       000013A8  CBREAK     000011B6
                                       CBREAKH    000011D4  CWAIT      00001192
                                       AINIT      000011F0  ARST       00001200
                                       ARWT       00001204  P1RD       0000120C
                                       PWRT       0000139C  P2RD       00001334
                                       ACLO       00001208
PASIO       8      000013F2  00001685  .PCLO      00001502  .PAFI      0000152E
                                       .PIFD      000013F2  .PRDBUF    000015B2
                                       .PRST      00001456  .PRWT      000014C0
                                       .PWRTBUF   0000163C
PRINT       8      00001686  000016BD  PRNCHR     00001686
TRAPS       8      000016BE  0000184F  .PADDRER   00001826  .PVBUSER   00001808
                                       .PVCHKI    000017EA  .PVTRAPD   000017EA
                                       .PVTRAPE   000016BE  .PVTRAPV   000017D0
                                       .PVZDIV    000017B6
ALSTS       8      00001850  0000187B  .PALSTS    00001850  .PALSTSL   00001852
WLN         8      0000187C  0000188B  .PWLN      0000187C
WRSWRV      8      0000188C  000018D5  .PWRS      00001896  .PWRV      0000188C
LBLKS       8      000018D6  000018E7  .PLBLKS    000018D6
IWPTR       8      000018E8  000018FB  .PIWPTR    000018E8
LDC         8      000018FC  00001925  .PLDCS     000018FC  .PLDCV     00001900
EXAMPLE     9      00001926  00001973  .PMAIN     00001926


Segment SEG2: 00001A00 00002EFF 0,15
Module      S   T  Start     End       Externally Defined Symbols

TABLE       0      00001A00  00001A1B
EXAMPLE    15      00001A1C  00002E1B  .PZMAIN    00002E1A
```

Table of Externally Defined Symbols:

Name	Address	Module	Displ	Sect	Seg	Library	Input
.PADDRER	00001826	TRAPS	00000168	8	SEG1	PASCALIB.RO	
.PAFI	0000152E	PASIO	0000013C	8	SEG1	VM01PLIB.RO	
.PALSTS	00001850	ALSTS	00000000	8	SEG1	PASCALIB.RO	
.PALSTSL	00001852	ALSTS	00000002	8	SEG1	PASCALIB.RO	
.PCLO	00001502	PASIO	00000110	8	SEG1	VM01PLIB.RO	
.PIFD	000013F2	PASIO	00000000	8	SEG1	VM01PLIB.RO	
.PINI	00001000	INIT	00000000	8	SEG1	VM01PLIB.RO	
.PIWPTR	000018E8	IWPTR	00000000	8	SEG1	PASCALIB.RO	
.PLBLKS	000018D6	LBLKS	00000000	8	SEG1	PASCALIB.RO	
.PLDCS	000018FC	LDC	00000000	8	SEG1	PASCALIB.RO	
.PLDCV	00001900	LDC	00000004	8	SEG1	PASCALIB.RO	
.PLJSR	0000108C	INIT	0000008C	8	SEG1	VM01PLIB.RO	
.PMAIN	00001926	EXAMPLE	00000000	9	SEG1		EXAMP2 .RO
.PRDBUF	000015B2	PASIO	000001C0	8	SEG1	VM01PLIB.RO	
.PRST	00001456	PASIO	00000064	8	SEG1	VM01PLIB.RO	
.PRWT	000014C0	PASIO	000000CE	8	SEG1	VM01PLIB.RO	
.PSTART	0000115E	START	00000000	8	SEG1	VM01PLIB.RO	
.PTABLE	00001094	TABLE	00000000	8	SEG1	VM01PLIB.RO	
.PVBUSER	00001808	TRAPS	0000014A	8	SEG1	PASCALIB.RO	
.PVCHKI	000017EA	TRAPS	0000012C	8	SEG1	PASCALIB.RO	
.PVTRAP1	00001098	TABLE	00000004	8	SEG1	VM01PLIB.RO	
.PVTRAPD	000017EA	TRAPS	0000012C	8	SEG1	PASCALIB.RO	
.PVTRAPE	000016BE	TRAPS	00000000	8	SEG1	PASCALIB.RO	
.PVTRAPV	000017D0	TRAPS	00000112	8	SEG1	PASCALIB.RO	
.PVZDIV	000017B6	TRAPS	000000F8	8	SEG1	PASCALIB.RO	
.PWLN	0000187C	WLN	00000000	8	SEG1	PASCALIB.RO	
.PWRS	00001896	WRSWRV	0000000A	8	SEG1	PASCALIB.RO	
.PWRTBUF	0000163C	PASIO	0000024A	8	SEG1	VM01PLIB.RO	
.PWRV	0000188C	WRSWRV	00000000	8	SEG1	PASCALIB.RO	
.PZMAIN	00002E1A	EXAMPLE	000013FE	15	SEG2		EXAMP2 .RO
ACLO	00001208	IO	00000076	8	SEG1	VM01PLIB.RO	
AINIT	000011F0	IO	0000005E	8	SEG1	VM01PLIB.RO	
ARST	00001200	IO	0000006E	8	SEG1	VM01PLIB.RO	
ARWT	00001204	IO	00000072	8	SEG1	VM01PLIB.RO	
AWRT	000013A8	IO	00000216	8	SEG1	VM01PLIB.RO	
CBREAK	000011B6	IO	00000024	8	SEG1	VM01PLIB.RO	
CBREAKAD	000010A0	TABLE	0000000C	8	SEG1	VM01PLIB.RO	
CBREAKB	0000109C	TABLE	00000008	8	SEG1	VM01PLIB.RO	
CBREAKH	000011D4	IO	00000042	8	SEG1	VM01PLIB.RO	
CBREAKHB	000010A4	TABLE	00000010	8	SEG1	VM01PLIB.RO	
CWAIT	00001192	IO	00000000	8	SEG1	VM01PLIB.RO	
CWAITAD	000010AC	TABLE	00000018	8	SEG1	VM01PLIB.RO	
CWAITB	000010A8	TABLE	00000014	8	SEG1	VM01PLIB.RO	
CWAITCH	000010B0	TABLE	0000001C	8	SEG1	VM01PLIB.RO	
DDTLINK	00001094	TABLE	00000000	8	SEG1	VM01PLIB.RO	
DEBUG	000010C2	TABLE	0000002E	8	SEG1	VM01PLIB.RO	
EOF	000010B2	TABLE	0000001E	8	SEG1	VM01PLIB.RO	
ILL	0000002C	ILL			SEG0	VM01PLIB.RO	
MEMBEG	000010C6	TABLE	00000032	8	SEG1	VM01PLIB.RO	
MEMEND	000010CA	TABLE	00000036	8	SEG1	VM01PLIB.RO	
OTH$DV	000010BE	TABLE	0000002A	8	SEG1	VM01PLIB.RO	
P1RD	0000120C	IO	0000007A	8	SEG1	VM01PLIB.RO	
P2RD	00001334	IO	000001A2	8	SEG1	VM01PLIB.RO	
PRNCHR	00001686	PRINT	00000000	8	SEG1	VM01PLIB.RO	
PWRT	0000139C	IO	0000020A	8	SEG1	VM01PLIB.RO	
START	000010B6	TABLE	00000022	8	SEG1	VM01PLIB.RO	
STD$DV	000010BA	TABLE	00000026	8	SEG1	VM01PLIB.RO	
TRP1HNDL	0000116E	TRAP1	00000000	8	SEG1	VM01PLIB.RO	

Unresolved References: None

Multiply Defined Symbols: None

```
Lengths (in bytes):

     Segment      Hex          Decimal

      SEG1     00000A00           2560
      SEG2     00001500           5376
Total Length   00002F00           7936


   No Errors
   No Warnings

S-record module has been created.
```

CHAPTER 9

RUNTIME INTERFACE FOR NON-VERSAdos SYSTEMS

9.1 GENERAL

M68000 Family Pascal is a powerful programming tool for the MC68000/MC68010 microprocessors. It includes runtime input/output (I/O) routines which interact with an operating system or monitor program. There are four versions of the runtime library for M68000 Family Pascal: one is for VERSAdos in a Memory Management Unit environment (EXORmacs or VME/10); one is for VERSAdos in a non-MMU environment (VM01, VM02, or MVME110); one is for RMS68K on VM01, VM02, or MVME110; and one is for VM01 with VERSAbug. This chapter describes these routines and their interaction with VERSAdos, RMS68K, and VERSAbug.

9.2 USER ADAPTATION

As supplied, the Pascal runtime routines for VERSAdos or RMS68K programs depend upon the availability of VERSAdos or RMS68K; the routines that are operating independent from the system -- for the VM01 system -- depend upon VERSAbug. However, Pascal programs can be modified for use on M68000 family systems not using VERSAdos or RMS68K, or for VERSAmodules not using VERSAbug.

9.2.1 VERSAdos Adaptation

VERSAdos performs two classes of functions for Pascal: input and output functions, and system resource allocation functions. Typically, input and output functions occur continuously during execution of the Pascal program. These functions include creation and deletion of files, assignment of files and devices, and actual input and output of information. The resource allocation functions occur at program startup. Resource allocation functions include specification of trap vector addresses and allocation of additional memory.

Due to the layered structure of Pascal programs, as depicted in Figure 9-1, there are two ways to adapt Pascal programs to non-VERSAdos systems. The layered structure results from the Pascal programs calling runtime routines which, in turn, call VERSAdos functions. Thus, in order to run a Pascal program on a non-VERSAdos system, either the VERSAdos functions can be simulated or the runtime routines which require VERSAdos can be replaced. Paragraphs 9.3 and 9.4, respectively, describe these adaptations. Refer also to the VERSAdos Data Management Services and Program Loader User's Manual for additional information on VERSAdos functions.

When a Pascal program will be doing disk I/O to an operating system other than VERSAdos, simulating the VERSAdos functions would probably be the better approach. In this case, a user-supplied program intercepts the calls to VERSAdos, converts them to appropriate calls to the new host operating system, and then converts the status and other information returned from the host system to the appropriate VERSAdos format. The user-supplied VERSAdos simulation program then becomes an additional layer in the Pascal program structure, as shown in Figure 9-2.

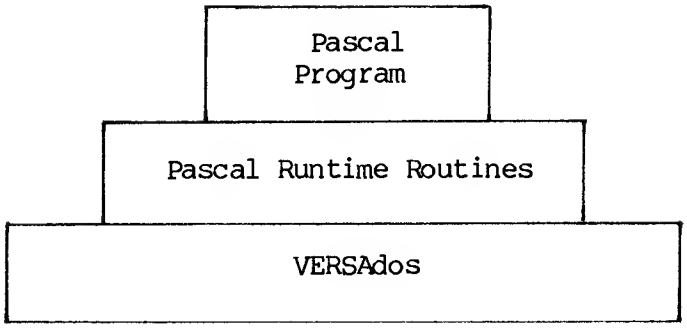


FIGURE 9-1. Layered Structure of Pascal Programs

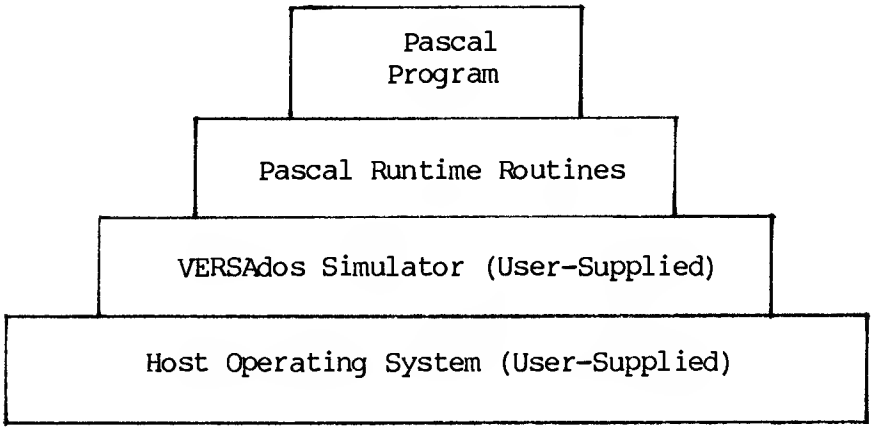


FIGURE 9-2. Pascal Program Structure with VERSAdos Simulation

When the Pascal program requires only relatively simple I/O, such as to a terminal, modifying the appropriate Pascal runtime routines is probably the better method. Here, the user writes routines that meet the interface specifications for runtime routines, then uses them in place of the corresponding Motorola-supplied routines. This approach is also useful if the host operating system does not lend itself to easy simulation of the VERSAdos functions. The Pascal program structure shown in Figure 9-3 results from this approach.

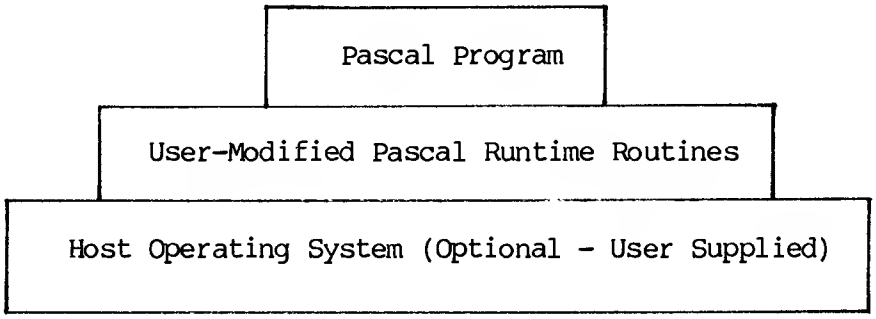


FIGURE 9-3. Pascal Program Structure with User Modified Runtime Routines

9.2.2 VERSAmodule 01 (with VERSAbug) Adaptation

Pascal is adapted for VERSAmodule 01 by means of VERSAmodule 01 runtime routines and I/O routines (Figure 9-4). However, user-supplied I/O routines could alternatively be supplied (Figure 9-5). Refer to paragraph 9.7 for information relevant to modifying VERSAmodule 01 routines.

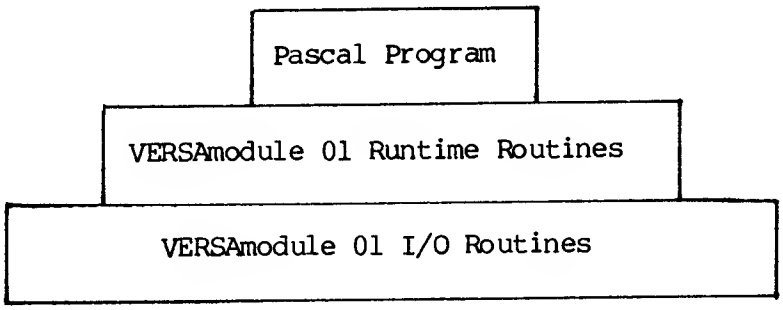


FIGURE 9-4. VERSAmodule 01 Pascal Program Structure #1

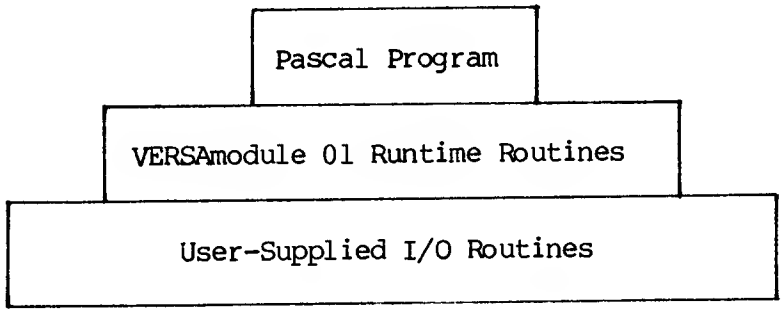


FIGURE 9-5. VERSAmodule 01 Pascal Program Structure #2

9.3 VERSAdos EMULATION

A basic knowledge of the purpose of a disk operating system (DOS) will aid in understanding functions provided by VERSAdos. A DOS manages disk resources by keeping a directory of names of files on the disk. This permits the DOS to access a file by name. Additionally, a DOS must maintain information about which areas of the disk are in use, so that new files may be added to the disk, and files already on disk may be expanded. It updates this information as files are deleted from the disk so that the space no longer used is made available.

Some disk operating systems, including VERSAdos, manage all I/O functions -- not just disk I/O. This results in I/O independent programs. For example, the program does not care if its output goes to a line printer or a disk file. The DOS takes care of all the details of data transfer. The program tells the DOS at execution time where the output is to go. Input operations have the same flexibility.

Refer to the VERSAdos Data Management Services and Program Loader User's Manual for additional information on input/output functions of VERSAdos, and to the VERSAdos System Facilities Reference Manual for file name formats.

VERSAdos utilizes logical unit numbers (LUN's) to provide device independent I/O. With this technique, a program directs its I/O through various logical units, each identified by a unique number. For example, a program might obtain its input data from LUN 1, write its output to LUN 2, and send error messages to LUN 3. Assignment of each of the logical units to a file or device would occur during program execution.

Optionally, the assignment can be made just before the program is executed. The LUN assigned is determined by the position of the file identifier in the Pascal PROGRAM statement, ignoring the identifiers INPUT and OUTPUT and their corresponding LUN's. The default LUN for the standard file identifier input is 5. The default LUN for the standard file identifier output is 6.

LUN's start with 1. The runtime routines limit the maximum LUN to 31. When a LUN is assigned to a file identifier at execution time, the next available LUN, beginning at 1, is used.

VERSAdos subdivides the I/O functions into file handling services (FHS, paragraph 9.3.1) and input output services (IOS, paragraph 9.3.2). The functions supplied by file handling services include: allocate, assign, retrieve attributes, delete, and close. These functions are associated with creating files and assigning files to logical units. The input/output services functions used by the runtime routines are: input, output, and rewind.

The runtime routines support two types of VERSAdos files: contiguous and sequential. Sequential files may have fixed length or variable length records. Sequential files with variable length records may be ASCII formatted, in which case VERSAdos compresses strings of multiple spaces to one byte when writing the file. VERSAdos also expands the space compression when reading the file. Variable length records may be written and read without ASCII formatting. Fixed length records do not permit ASCII formatting. Pascal text files are sequential.

Contiguous files have a fixed record length of 256 bytes, and are primarily used for VERSAdos load modules. This is the same size as a VERSAdos disk logical sector. Contiguous files do not support ASCII formatting. To enhance the load time, these files are actually allocated contiguous space on the disk. Because of this, the size of the contiguous file must be specified when the file is created. After a contiguous file has been created, its size cannot be changed. A VERSAdos contiguous file that has just been created (allocated) cannot be read until the entire file has been written.

Sequential files are not necessarily allocated contiguous disk space. The smallest amount of space for a sequential disk file that must be physically contiguous is called a data block. The smallest size and the default size of a data block is four sectors. VERSAdos does not permit records to straddle data block boundaries. An additional block of a sequential file, called a File Access Block (FAB), contains pointers to all of the data blocks of the file. For more information about FAB's, see the VERSAdos System Facilities Reference Manual.

Both contiguous and sequential files permit several types of access, including: random access based on record number, previous record, current record, and next record. The runtime routines utilize only the next record type, which is sequential access.

Due to the multitasking nature of VERSAdos, it has an active file protection. The active protection is specified when a program attaches to (assigns) a device or file. It not only specifies what type of I/O will be done to the file, but whether or not other programs may do I/O to the file. The types of active protection are: public read, exclusive read, public write, exclusive write, public read-public write, public read-exclusive write, exclusive read-public write, exclusive read-exclusive write. A public permission assignment permits other public permission assignments of the same type. An exclusive assignment prohibits other assignments of the same type. Compatible access permission assignments are shown in Table 9-1.

TABLE 9-1. Compatible Access Permissions

Existing Access Permission	PR	ER	PW	EW	PRPW	PREW	ERPW	EREW
Public Read	X		X	X	X	X		
Exclusive Read			X	X				
Public Write	X	X	X		X		X	
Exclusive Write	X	X						
Public Read-Write	X		X		X			
Public Read, Exclusive Write	X							
Exclusive Read, Public Write			X					
Exclusive Read-Write								

X = Other assignment allowed.

The default access permission for a temporary file is public read-exclusive write. For everything else, the default access permission is public read for a reset and public write for a rewrite.

Pascal runtime treats text and non-text files somewhat differently in terms of default file specifications. Table 9-2 shows the default I/O specifications for text and non-text files. Some parameters are effective only when the file is allocated. For example, specifying the data block size when the file is assigned has no affect, since the data block size was fixed when the file was created. The Pascal program can override the file defaults by specifying appropriate options in the resource name string of the reset and rewrite procedures. The resource name string options are listed in Table 9-3.

TABLE 9-2. Default File Specifications

	TEXT	NON-TEXT
File Type	Sequential	Sequential
Record Length	Variable	Fixed to component size
ASCII Format	Yes	No
Data Block Size	0=default (4)	0=default (4)
FAB size	0=default (1)	0=default (1)

TABLE 9-3. Resource Name String Options

OPTION	MEANING
B	Binary I/O, record length = component size
C=	Contiguous file and its size
D=	Data block size
F=	Size of file access block
R	Public read access permission
-R	Exclusive read access permission
W	Public write access permission
-W	Exclusive write access permission

9.3.1 File Handling Services (FHS)

The File Handling Services (FHS) functions are called by a TRAP #3 instruction. (Refer to Table 9-4.) The sequence of instructions used to call FHS functions is:

```
MOVE.L    #PARAM,A0
MOVE.W    #FHSFUN,(A0)
TRAP      #3
```

PARAM is the starting address of a 40-byte FHS parameter block. The address will always be on a word boundary. FHSFUN is a word value which identifies the FHS function being requested.

On return from the trap, the only processor registers modified are D0 and the condition code register. D0 contains a byte value which is the status returned from the function. A status value of zero indicates that no error occurred. If non-zero, the status value indicates the type of error that has occurred. The Z bit of the condition code register is set according to the value of the status byte in D0. The status byte is also returned in a byte of the parameter block. Refer to the VERSAdos Data Management Services and Program Loader User's Manual for details about FHS functions.

Table 9-5 lists the FHS functions called by each runtime routine.

The FHS option "allocate shared buffer" is not requested by runtime routines.

User-specified attributes are not used by runtime routines.

TABLE 9-4. Traps Used by Pascal

TRAP	FUNCTION
TRAP #1	Executive functions
TRAP #2	IOS calls
TRAP #3	FHS calls
TRAP #4	Error messages
TRAP #13	Runtime range checking/errors
TRAP #14	Error/program termination

TABLE 9-5. FHS Functions Called by Each Runtime Routine

	CALLED BY:						
FHS FUNCTION	IFD	RST	RWT	AFI	READBUF	WRTBUF	CLOSE
Delete			X				
Close	X	X	X	X	X		X
Assign		X	X		X		
Allocate			X				
Retrieve Attributes	X						

9.3.2 Input Output Services (IOS)

The Input Output Services (IOS) functions are called by a TRAP #2 instruction. The sequence of instructions used to call IOS functions is:

```
MOVE.L    #PARAM,A0
MOVE.W    #IOSFUN,(A0)
TRAP      #2
```

PARAM is the starting address of a 28-byte IOS parameter block. The address will always be on a word boundary. IOSFUN is a word value which identifies the IOS function being requested. On return from the trap, the only processor registers modified are D0 and the condition codes register. D0 contains a byte value which is the status returned from the function. A status value of zero indicates that no error occurred. A non-zero value indicates that an error occurred. The value indicates the type of the error. The Z bit of the condition codes register is set according to the value of the status byte in D0. The status is also returned in a byte of the parameter block. Refer to the VERSAdos Data Management Services and Program Loader User's Manual for details about IOS functions.

The IOS functions used by the runtime routines are as follows:

Read
Write
Rewind

Table 9-6 lists the IOS data transfer options used by the runtime routines. Unless the RESET or REWRITE resource name string specifies the binary option, ASCII formatting is used for text files. All non-text files utilize binary formatting. The runtime routines do not write indexed sequential files. When indexed sequential files are read, the runtime routines do not expect to receive the record key. Also, the runtime routines do not use the format disk option.

TABLE 9-6. Runtime Routine IOS Data Transfer Options

OPTIONS	DEFAULT VALUE
ASCII/BINARY	- ASCII formatting for text files - Binary formatting for non-text files
WAIT/PROCEED	- Wait
FORMATTED/IMAGE	- Formatted
BREAK NOTIFICATION	- Shared notification
SUPPRESS ECHO	- Do not suppress echo
RECORD/BLOCK ACCESS	- Record access
LOGICAL RECORD/RANDOM KEY ACCESS	- Logical record number
RETURN KEY WITH RECORD	- Do not return key
COMPLETION/SERVICE ADDRESS	- Normal return
INPUT FORMATTED/IMAGE	- Formatted
PRIMARY/SECONDARY MEMORY MAP	- Primary memory map
LOGICAL ACCESS/POSITION	- Access next record

Table 9-7 lists the IOS functions called by each runtime routine. The runtime routines are described in paragraph 9.4. The exact action of the IOS functions depends on the file type being operated on -- contiguous or sequential.

TABLE 9-7. IOS Functions Called by Each Runtime Routine

	CALLED BY:						
IOS FUNCTION	IFD	RST	RWT	AFI	READBUF	WRTBUF	CLOSE
Read					X		
Write		X				X	X
Rewind		X					

9.3.3 Executive Functions

VERSAdos permits multitasking. To do this, VERSAdos contains executive functions which allocate system resources and provide task protection. A four-segment memory management unit (MMU) on the EXORmacs or VME/10 processor board provides memory protection between and within tasks. Additional protection is provided by all non-executive tasks running in the user mode of the MC68000 or MC68010 processor.

A TRAP #1 instruction is used to request executive functions. Before executing the trap instruction, a value indicating the desired function is placed in D0, and the address of a parameter block (if required) is placed in A0. On return, D0 contains a status code, and the condition code register is set to reflect the contents of D0. A value of zero indicates that no errors occurred. Additionally, some functions return information in A0. Those that do not return information in A0 leave A0 unchanged. All other registers are returned unchanged. Table 9-8 lists the executive functions used by the runtime routines, and the runtime routine that calls each. The instruction sequence used to call executive functions is:

```
MOVE.L    #DIRNUM,D0
MOVE.L    #PARAM,A0
TRAP      #1
```

DIRNUM is the directive number which indicates what executive function is being requested. PARAM is the address of the first byte of the parameter block. The parameter block must start on a word boundary. Refer to the M68000 Family Real-Time Multitasking Software User's Manual for details about executive functions.

TABLE 9-8. Executive Functions Used by Runtime Routines

	USED BY:	
EXECUTIVE FUNCTION	INITIALIZATION	TRAP HANDLER
Receive segment attributes	X	
Get segment	X	
Receive exceptions	X	
Receive traps	X	
Abort	X	X
Terminate		X

The two runtime routines that call executive functions are Initialization and TRAP #14 Handler. Initialization sets up the runtime environment for the Pascal program. To do this, it requests information about the area of memory used for the stack and heap. It might also request additional memory for the stack-heap, depending on the user-specified command line options. Initialization also requests that the executive let Pascal runtime routines handle certain exceptions and traps. One of the traps handled by the runtime routines is TRAP #14. It is called any time an error is detected, and provides for an orderly termination of the program. It is also called at the end of the program. Initialization aborts itself if it detects an error before it has requested runtime routine service of the TRAP #14 vector. Another trap, TRAP #13, is used by the runtime routines to indicate failure during range checking. Initialization and trap handling are described more completely in paragraphs 9.5 and 9.6.

Initialization uses the "receive segment attributes" function to obtain the logical beginning and ending addresses of the stack/heap memory segment.

When the Z option is used, initialization uses the "get segment" executive function to obtain an additional memory segment to extend the size of the stack-heap area.

The exceptions that the runtime routines receive are: bus error, address error, zero divide, CHK instruction, and TRAPV instruction.

The runtime routines service TRAP #13 and TRAP #14.

The trap handler runtime routine calls the VERSAdos error message routine using TRAP #4.

9.4 I/O ROUTINE REPLACEMENT

One way to run Pascal programs on non-VERSAdos systems is to replace the specific VERSAdos functions required by the Pascal runtime routines. This is described in paragraph 9.3. Another way is to replace the Pascal runtime routines and trap handlers. Paragraph 9.4 describes the replacement of the Pascal runtime I/O routines so that they may be tailored to the specific I/O requirements of the host operating system. Paragraph 9.5 describes the replacement of the Pascal initialization runtime routine. Paragraph 9.6 describes the replacement of the exception and trap handlers. Paragraph 9.7 describes differences for the VERSAmodule 01.

Runtime I/O is provided by five routines which the Pascal program calls directly, plus two routines which are called indirectly. The direct routines are AFI, CLO, IFD, RST, and RWT; the indirect routines are RDBUF and WRTBUF.

- AFI (Assign File Identifier to resource name string) is called immediately before a reset or rewrite if the reset or rewrite contains a resource name string.

- CLO (Close) is called at the end of a procedure containing local files in order to close them, and at the end of the program to close any global files. It is called once for each file to be closed.

- IFD (Initialize File Descriptor) is called at the start of the main program to initialize a data area for each global file, and at the start of each procedure containing local files to initialize a data area for each one. It is called once for each declared file identifier and for the file identifiers input and output if they are included in the program statement.

- RST (Reset) is called for each reset statement in the program. If the file identifier input is included in the program statement, a call to RST is automatically generated for it by the compiler.

- RWT (Rewrite) is called for each rewrite statement in the program. If the file identifier output is included in the program statement, a call to RWT is automatically generated for it by the compiler.

- RDBUF (Read Buffer) is called by a number of different runtime routines.

- WRTBUF (Write Buffer) is called by a number of different runtime routines.

The logical sequence of calls to the runtime I/O routines is as follows:

a. IFD is called to allocate and initialize the control tables required for the file.

b. If the reset or rewrite contains a resource name string, AFI is called immediately before the reset or rewrite. AFI modifies the file control tables as required by the resource name string.

c. RST or RWT is called, depending on the Pascal statement, for reset or rewrite, respectively.

d. The I/O is done. RDBUF is called to read data from a file that has been reset; WRTBUF is called to write data to a file that has been rewritten.

e. Finally, CLO closes the file.

To allow runtime routine changes without requiring recompilation of programs, the runtime routines are accessed by external references which are resolved at linkage edit time. Therefore, to link to runtime routines other than those in the standard Pascal library, a user library must be built. The user must issue the proper linkage editor commands to read the user library before the standard library (and after the Pascal program). This will be sufficient to get the user-supplied runtime routines linked, because the Pascal program will make external references to them.

However, the Pascal program does not make direct external references to RDBUF and WRTBUF. These routines are referenced by various routines in the standard library. To use the user's version of these routines, one of the user's replacement runtime routines must include external references to them, and the routines must follow the reference to them in the user's library. It is not necessary for the routine that contains the external references to RDBUF and WRTBUF to make any use of them.

The external label used for a runtime or other routine is the runtime or routine name preceded by '.P'. Table 9-9 contains a list of the external references that must be replaced to be independent of VERSAdos.

TABLE 9-9. External References that Cause VERSAdos Calls

EXTERNAL REFERENCE	ROUTINE
.PAFI	Assign file identifier to resource name string
.PCLO	Close
.PIFD	Initialize file descriptor
.PRST	Reset
.PRWT	Rewrite
.PRDBUF	Read buffer
.PWRTBUF	Write buffer
.PLJSR	Initialization (long JSR)

After the replacement library has been built, by merging the relocatable object modules in the proper order into one file, the replacement library can be invoked by the following VERSAdos command:

```
=LINK <myprog>;L=<mylib>
```

where <myprog> is the user's relocatable object module of a Pascal program, and <mylib> is the user's replacement library.

The linker first reads the file <myprog>, then loads the required modules from the user's library file <mylib>. Finally, it loads the required modules from the default Pascal library file PASCALIB. The same effect can be achieved by entering the command LIB <mylib> before the END command and after the INPUT <myprog> command when using the linker in the interactive mode.

The runtime I/O routines and a number of other runtime routines make use of a global data area called the Runtime Maintenance Area (RMA). Table 9-10 shows the layout of the RMA. A pointer to the start of the RMA is kept in A5 during the execution of the Pascal program.

Except for the array of 2-byte integers, the area from the loading/initiating task name up to the stack/heap segment break address holds information required for various runtime I/O and VERSAdos I/O and executive functions. The stack/heap segment break address is used to keep file descriptors from straddling a segment break caused by adding a memory segment to the stack/heap. VERSAdos requires the file descriptor to reside completely within one memory segment.

When replacing the standard runtime I/O routines, the user can also redefine much of the RMA. The area of the RMA from loading/initiating task name up to smallest heap pointer value may be redefined if the runtime I/O routines are replaced.

The remaining areas of the RMA are integral to the operation of Pascal and may not be altered.

TABLE 9-10. Pascal Runtime Maintenance Area (VERSAdos)

Offsets from A5 hex	Offsets from A5 decimal	
0	0	Statement Counter
4	4	Heap Pointer
8	8	Display Level 0
C	12	Display Level 1
10	16	Display Level 2
14	20	Display Level 3
18	24	Display Level 4
1C	28	Display Level 5
20	32	Display Level 6
24	36	Display Level 7
28	40	Loading/Initiating Task Name
2C	44	Session Number
30	48	LU's Passed/In Use
34	52	Chain/batch diagnostic register | First 2 chars of term id.
38	56	Second 2 chars of term id. | First 2 chars of default vol.
3C	60	Second 2 chars of default vol. | Default User Number
40	64	Default Catalog
48	72	Command Line Length | Command Line
		(162 characters - bytes)
EC	236	Array ['A'..'Z'] of 2-byte Integer (52 bytes)
120	288	Unused | # of fields in command line
124	292	# of files in I= | cur file in I= | offset to field | offset to cur file
128	296	# of files in O= | cur file in O= | offset to field | offset to cur file
12C	300	# of files in field 1 | cur file in field 1 | offset to field | offset to cur file
12C+(n-1)x4	300+(n-1)x4	⋮
168	360	# of files in field 16 | cur file in field 16 | offset to field | offset to cur file
16C	364	Stack/Heap Segment Break Address
170	368	Smallest Heap Pointer Value
174	372	Runtime Error Routine Storage (72 bytes)
1BC	444	

Paragraphs 9.4.1 through 9.4.7 describe the function of the standard runtime I/O routines. Register formats upon entry to and exit from the routines are listed in Table 9-11. The contents of registers not listed are indeterminate.

TABLE 9-11. Register Formats for I/O Subroutines (VERSAdos)

SUB-ROUTINE	TIME	REGISTER	CONTENTS
AFI	on entry	A3.L	Address of long JSR routine
		A5.L	Global variable pointer
		A6.L	Local variable pointer
		A7.L	Stack pointer
		(A7)+0	Return address (long word)
		(A7)+4	String length (word)
		(A7)+6	String (even string length (s.l.) bytes)
		(A7)+6+even s.l.	Address of file pointer
		(A7)+10+even s.l.	Previous stack contents
	on exit	A0.L	Address of file pointer
		A3.L-A6.L	Preserved
		A7.L	Points to previous stack contents
		D0.W	Status, if error
		D7.L	Return address from AFI if error
IFD	on entry	A0.L	Address of file pointer
		A3.L	Address of long JSR routine
		A5.L	Global variable pointer
		A6.L	Local variable pointer
		A7.L	Stack pointer
		D0.B (low order 8 bits)	File's position in program statement
		D1.W (low order 16 bits)	Initial file status
		D2.W	Component size
	on exit	A3.L-A6.L	Preserved
		A7.L	Points to first byte of allocated space
		D0.W	Status, if error
		D7.L	Return address from IFD if error
RST and RWT	on entry	A0.L	Address of file pointer
		A3.L	Address of long JSR routine
		A5.L	Global variable pointer
		A6.L	Local variable pointer
		A7.L	Stack pointer
	on exit	A3.L-A6.L	Preserved
		D0.W	Status, if error
		D7.L	Return address from RST or RWT if error

TABLE 9-11. Register Formats for I/O Subroutines (VERSAdos) (cont'd)

SUB-ROUTINE	TIME	REGISTER	CONTENTS
RDBUF	on entry	A0.L	Address of file pointer
		A2.L	Component pointer
		A5.L	Global variable pointer
		A6.L	Local variable pointer
		A7.L	Stack pointer
	on exit	A0.L,A1.L	Preserved
		A2.L	0 if EOF; else beginning address of record buffer
		A3.L-A6.L	Preserved
		A7.L	Points to previous stack contents
		D0.W	Status, if error
		D7.L	Return address from RDBUF if error
WRTBUF	on entry	A0.L	Address of file pointer
		A1.L	Component pointer
		A2.L	Address of file parameter block
		A5.L	Global variable pointer
		A6.L	Local variable pointer
		A7.L	Stack pointer
	on exit	A0.L	Preserved
		A1.L	Beginning address of record buffer
		A2.L-A6.L	Preserved
		A7.L	Points to previous stack contents
		D0.W	Status, if error
		D7.L	Return address from WRTBUF if error
CLO	on entry	A0.L	Address of file pointer
		A3.L	Address of long JSR routine
		A5.L	Global variable pointer
		A6.L	Local variable pointer
		A7.L	Stack pointer
	on exit	A3.L-A6.L	Preserved
		D0.W	Status, if error
		D7.L	Return address from CLO if error

NOTE: If one of these routines encounters an error, it puts an error status in D0 and the address it would have returned to, had the error not occurred, in D7. It then executes a TRAP #14, which is followed in memory by a word indicating the type of error. Paragraph 9.8 describes the TRAP #14 error handler requirements.

9.4.1 Assign File (AFI)

Assign file (AFI) updates the file parameters as specified in a resource name string contained on the stack. AFI is called immediately before any RWT or RST which includes a resource name string. The RMA is accessed using positive offsets from A5. The contents of registers A3 and A6 are generally not required by AFI. However, the contents of A3, A5, and A6 must be preserved through the AFI routine -- that is, their contents on exit from AFI must be the same as their contents were on entry to AFI.

The stack contains several items of importance to AFI. At the top of the stack, pointed to by A7, is the address to return control to on completion of AFI. The return address is stored as a long word. Next on the stack, at a positive offset of 4 from the contents of A7, is the length of the string passed to AFI. The string length is stored as a word. The string is stored on the stack next in an even number of bytes. If the string length is odd, an additional byte is padded at the end of the string on the stack to keep the stack aligned on even addresses. The first character of the string follows immediately after the string length word. The string is packed one character per byte.

After the string comes the address of the file pointer (Table 9-12).

TABLE 9-12. File Pointer

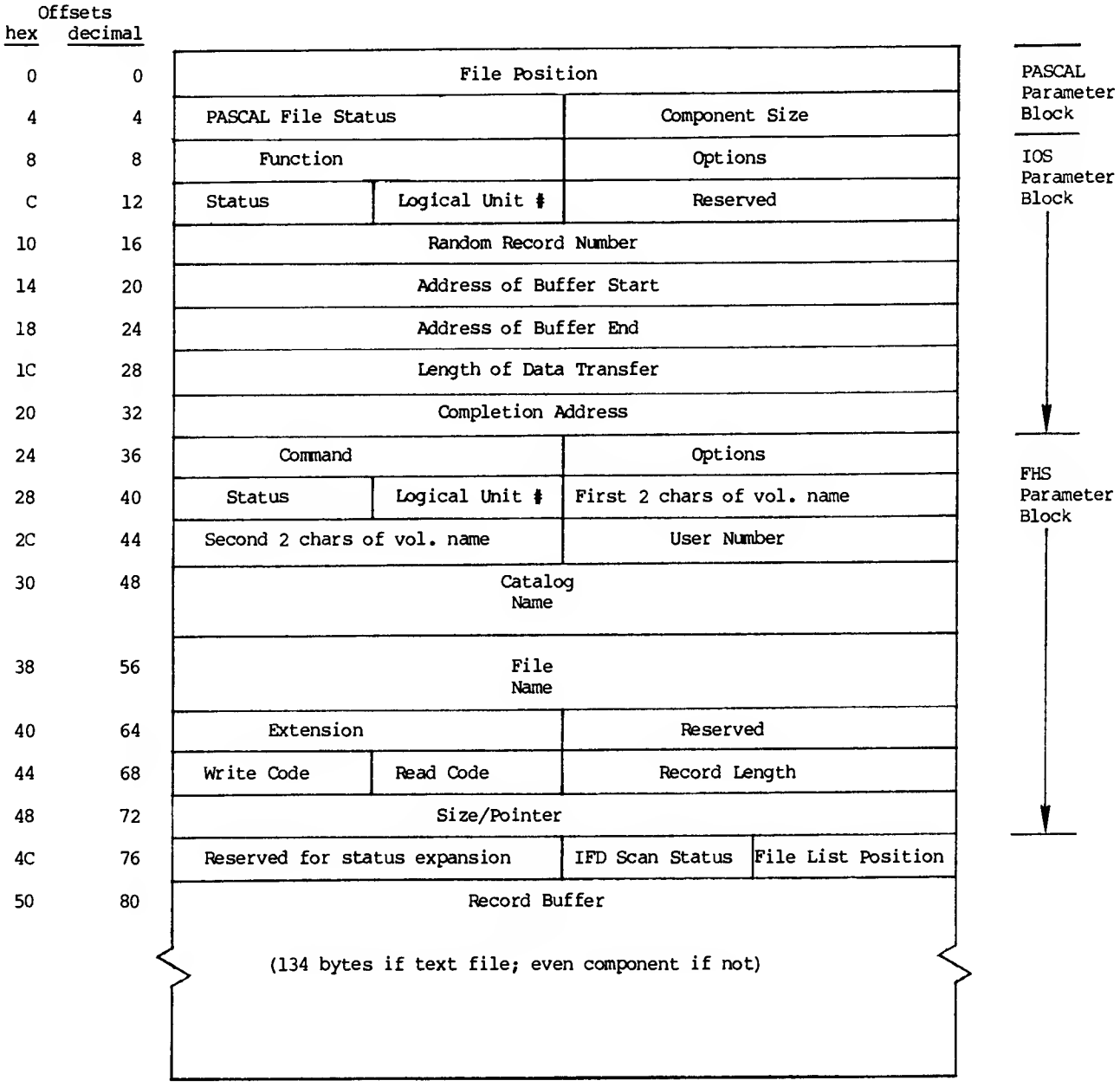
File Pointer Address	0	Current Component Pointer
File Pointer Address+4	4	File Parameter Block Pointer

The file pointer provides the Pascal program with a means of keeping track of the file I/O. The file pointer contains two addresses, each stored as a long word. The first address is the current component pointer. It is either 0 or points into the file's data buffer. On input from the file, it is 0 if the buffer contents are not valid, such as just after the reset statement. Otherwise, during input, it points to the next character to be read from the buffer. For output, the component pointer points to the next location in the buffer where a byte of data is to be put. The file parameter block pointer points to the parameter block created for the file by IFD.

The file parameter block created by the Motorola-supplied IFD routine is shown in Table 9-13. The first eight bytes of the parameter block contain information used by the Pascal program. Next comes the IOS parameter block. After that is the FHS parameter block. (Refer to the Data Management Services and Program Loader User's Manual for details about the FHS and IOS parameter blocks.) A long word containing information about the file name comes next. Finally, the file parameter block ends with the record buffer, which is 134 bytes long if the file is a text file. For a non-text file, the buffer length is the component size, rounded out by one byte if it is odd. Pascal programs require the address of the buffer end from the IOS portion and the first three parameters. The user's I/O procedures may redefine the rest of the file parameter block. The file position parameter is a long word containing the count of the current

component. The Pascal file status is a word that describes the file attribute. Table 9-14 defines the file status word. To gain access to these parameter blocks, pass the file pointer as a variable parameter to an assembly language subroutine. The stack will have the address of the file pointer which, in turn, will point to the file parameter block described in Table 9-13.

TABLE 9-13. Standard File Parameter Block



Pascal programs make use of bits 0 through 4 and 7 and 8 of the file status word. The other bit definitions are used by the Motorola-supplied runtime I/O routines, and may be redefined as required by the user's I/O routines. The component size word contains the number of bytes per component. The IFD scan status byte (Table 9-13) indicates which file name fields were found in the command line by IFD. The file list position gives the number of the position of the file name in the command line.

On exit from AFI, registers A3, A5, and A6 must contain the same values they had on entry to AFI. The stack pointer, A7, must point to the previous stack contents. Also, the address of the file pointer must be in A0. This was under the string on the stack. This is required because either RST or RWT is called immediately upon return from AFI.

If AFI detects an error, it executes a TRAP #14, which is followed in memory by a word indicating the type of error. The possible values of this word are described in paragraph 9.8 on handling traps. Some errors also return a status in D0. D7 should contain the address to return to from AFI when the TRAP #14 is executed.

TABLE 9-14. Pascal File Status Word Definition

BIT NUMBER	DESCRIPTION
0	Set to 1 if the file is standard output; otherwise, 0.
1	Set to 1 if the file is standard input; otherwise, 0.
2	Set to 1 for a text file; otherwise, 0.
3	Set to 1 for a local file; otherwise, 0.
4	Set to 1 for an indexed file; otherwise, 0.
5	Set to 0. Reserved for future use.
6	Set to 1 if the file is open; otherwise, 0.
7	Set to 1 if end of file; otherwise, 0.
8	Set to 1 if end of line; otherwise, 0.
9	Set to 1 if the file assignment was passed to the Pascal program; otherwise, 0.
10	Set to 1 if the access permission was changed by AFI; otherwise, 0.
11	Set to 1 if the file was specified in the command line; otherwise, 0.
12	Set to 1 if the file has been 'rewritten'; otherwise, 0.
13,14,15	Set to 0. Reserved for future use.

9.4.2 Initialize File Descriptor (IFD)

Initialize file descriptor (IFD) creates and initializes the file parameter block used by the Pascal program and the runtime I/O routines. It is called at the start of the main program, once for each globally declared file, including standard input and standard output if they are listed in the program statement. It is also called at the start of each procedure containing local files, once for each local file declared. The RMA is accessed using positive offsets from A5. The contents of registers A3 and A6 are generally not required by IFD. However, the contents of A3, A5, and A6 must be preserved through the IFD routine -- that is, their contents on exit from IFD must be the same as their contents were on entry to IFD.

D0 contains the byte value of the file's position in the PROGRAM statement. The position numbering begins at 1 and does not include the file identifiers input and output. The Motorola-supplied runtime I/O routines use this number to match up a command line field with a file identifier. D1 contains the initial file status. This is a subset of the file status word described in Table 9-14. The bits of the file status, as received in D1, that may be set are 0 through 5. A0 contains the address of the file pointer for this file. The file pointer is shown in Table 9-12 and described in the AFI section. However, on entry to IFD, the file parameter block does not exist.

One of the functions of IFD is to obtain space on the stack for the file parameter block, and to initialize it. Space on the stack is allocated by the .PALSTS runtime routine. It is XREF'ed in the IFD routine. On entry to it, D4 contains the number of bytes required. .PALSTS checks for stack/heap overflow before allocating the space on the stack. If stack/heap overflow would occur because of the allocation, then .PALSTS aborts the program with a stack/heap overflow error. Otherwise, it allocates the space on the stack and returns with A7, the stack pointer, pointing to the first byte of the allocated space. An even number of bytes is always allocated. If D4 is odd, then 1 is added to it to make it even. On return from .PALSTS, all registers except D4 and A7 are preserved. Note that before allocating space on the stack, a return address from IFD exists at the top of the stack. This can be removed from the stack and saved in a register before the allocate; then to return from IFD, a jump indirect through the register is executed.

Once IFD has the file parameter block allocated on the stack, it initializes the parameters. This involves saving D1 as the file status and D2 as the component size. Also, IFD sets up the buffer start and end addresses in the IOS portion of the parameter block, along with the other IOS and FHS parameters required. If the file is not local and the initialize routine finds a file name field in the command line for this file, then IFD fills in the file name from the command line. If the file is local, IFD sets up the file name field so that the operating system will generate a temporary file. VERSAdos will supply a unique file name when a temporary file is requested. IFD also puts the address of the file parameter block into the proper field of the file pointer.

On exit from IFD, A3, A5, and A6 are returned with their original contents. If IFD detects an error, it executes a TRAP #14. TRAP #14 is followed in memory by a word indicating the type of error. The possible values of this word are described in paragraph 9.8 on handling traps. Some errors also return a status in D0. D7 should contain the address to return to from IFD when the TRAP #14 is executed.

9.4.3 Close (CLO)

Close (CLO) closes a file assignment. Before closing the file assignment, CLO writes out anything left in the record buffer if the file was an output file. A file is an output file if the last RESET or REWRITE operation on it was REWRITE. This can be determined by having RWT set the output bit in the file status word and having RST reset the bit. Then, if the bit is set at CLO time, it is an output file. An output record buffer is empty if the current component pointer, contained in the file pointer, is equal to the beginning address of the record buffer. CLO should not attempt to de-allocate the file parameter block from the stack. The Pascal program will take care of that.

The RMA is accessed using positive offsets from A5. The contents of registers A3 and A6 are generally not required by CLO. However, on exit from CLO, the contents of A3, A5, and A6 must be the same as their respective contents were on entry to CLO. CLO is exited by executing the RTS instruction.

If CLO detects an error, it executes a TRAP #14, which is followed in memory by a word indicating the type of error. The possible values of this word are described in paragraph 9.8 on handling traps. Some errors also return a status in D0. D7 should contain the address to return to from CLO when the TRAP #14 is executed.

9.4.4 Reset (RST)

Reset (RST) opens a file assignment for read so that reading will start at the beginning of the file. RST permits the file to have been previously RESET or REWRITTEN when it is called. If the file was RESET last, it is just rewound. If the file was REWRITTEN last, then, if the record buffer is not empty, it is written to the file before the file is rewound for input. The CLO description tells how to determine if a file has been REWRITTEN and the record buffer is not empty. RST puts 0 in the current component pointer of the file pointer. This marks the buffer contents as not valid for the Pascal program. RST also puts 0 in the file position entry of the file parameter block.

The RMA is accessed using positive offsets from A5. The contents of registers A3 and A6 are generally not required by RST. However, on exit from RST, the contents of registers A3, A5, and A6 must be the same as their respective contents were on entry to RST. RST is exited by executing the RTS instruction.

If RST detects an error, it executes a TRAP #14, which is followed in memory by a word indicating the type of error. The possible values of this word are described in paragraph 9.8 on handling traps. Some errors also return a status in D0. D7 should contain the address to return to from RST when the TRAP #14 is executed.

9.4.5 Rewrite (RWT)

Rewrite (RWT) opens a file assignment for write so that writing will start at the beginning of the file. If the file already exists when RWT is called, then RWT deletes the existing file and creates a new one. RWT sets up the current component pointer to point to the beginning of the record buffer. RWT sets up the address of the buffer end of the IOS portion of the file parameter block. For a text file, the buffer end is the buffer start plus 131 bytes. A nontext file has a buffer end equal to the buffer start plus the component size. The buffer ending address, then, is equal to the starting address plus the buffer size minus one. RWT also puts 0 in the file position entry of the file parameter block.

The RMA is accessed using positive offsets from A5. The contents of registers A3 and A6 are generally not required by RWT. However, on exit from RWT, the contents of registers A3, A5, and A6 must be the same as their respective contents were on entry to RWT. RWT is exited by executing the RTS instruction.

If RWT detects an error, it executes a TRAP #14, which is followed in memory by a word indicating the type of error. The possible values of this word are described in paragraph 9.8 on handling traps. Some errors also return a status in D0. D7 should contain the address to return to from RWT when the TRAP #14 is executed.

9.4.6 Read Buffer (RDBUF)

Read Buffer (RDBUF) reads the next buffer of data from a file. It sets the end of file bit to 1 in the file status word of the file parameter block if there is no more data available from the file; otherwise, it is set to 0. If the file is a text file and if the length of the record read is 0, RDBUF sets the end of line bit; otherwise, it clears it. Also, for a text file, RDBUF puts an ASCII space character (decimal 32) at the end of the line in the buffer. RDBUG sets up the address of the buffer end in the IOS portion of the file parameter block if it is not end of file. For a text file, and because of the space character appended to the record, the buffer end address is equal to the buffer start address plus the length of the record transferred. For a nontext file, the buffer end address is equal to the buffer start address plus the length of the record transferred minus one.

The only register that should be modified by RDBUF on return is A2. The stack pointer, A7, will be modified due to the RTS instruction used to return from RDBUF, pulling the return address from the top of the stack. However, all of the other registers (D0 through D7, A0, A1, A3 through A6) should be preserved throughout RDBUF. To accomplish this, the registers needed may be saved by pushing them on the stack. They would then be restored by pulling them from the stack before returning from RDBUF. A value of 0 should be returned in A2 if RDBUF encounters the end of file. Otherwise, the beginning address of the record buffer should be returned in A2.

If RDBUF detects an error, it executes a TRAP #14, which is followed in memory by a word indicating the type of error. The possible values of this word are described in paragraph 9.8 on handling traps. Some errors also return a status in D0. D7 should contain the address to return to from RDBUF when the TRAP #14 is executed.

9.4.7 Write Buffer (WRTBUF)

Write Buffer (WRTBUF) writes the current buffer of data to a file. On entry to WRTBUF, A1 points one byte past the end of the record to be written. After writing the record, WRTBUF updates the end of buffer pointer in the IOS portion of the file parameter block in the same way that REWRITE initializes it. The buffer end address is equal to the buffer start address plus the buffer length minus one. The buffer length used for a text file is 132 bytes. The buffer length for a nontext file is the component size. The Motorola-supplied WRTBUF routine must restore the buffer end address, because it gets modified when the buffer is written to the file.

The only register that should be modified by WRTBUF on return is A1. The stack pointer, A7, will be modified due to the RTS instruction used to return from WRTBUF, pulling the return address from the top of the stack. However, all of the other registers (D0 through D7, A0, and A2 through A6) should be preserved through WRTBUF. To accomplish this, the registers needed may be saved by pushing them on the stack. They would then be restored by pulling them from the stack before returning from WRTBUF. The beginning address of the record buffer should be returned in A1.

If WRTBUF detects an error, it executes a TRAP #14, which is followed in memory by a word indicating the type of error. The possible values of this word are described in paragraph 9.8 on handling traps. Some errors also return a status in D0. D7 should contain the address to return to from WRTBUF when the TRAP #14 is executed.

9.5 INITIALIZATION UNDER VERSAdos (EXORmacs, VME/10, VM01, VM02, or MVME110)

The Motorola-supplied initialization routine accomplishes several functions, including:

a. Setting up the RMA.
b. Scanning the command line, if requested.
c. Allocating additional memory, if required.
d. Requesting trap and exception vectors.
e. Setting up processor registers for the Pascal program.
f. Calling the floating point initialization routine, if required.

Depending on what the user has done with the other I/O routines and the configuration of the system, some of the initialization functions can be simplified or even eliminated. For example, if the user's I/O routines do not require all of the RMA field, the initialization of the RMA can be simplified. If the user's system does not pass a command line to the Pascal program, there is no need for the initialization routine to contain code for scanning the command line. Most of the other initialization functions, except processor register initialization for the Pascal program, can be similarly reduced or eliminated, as permitted by the program's environment and I/O routines.

The parameters required by the Motorola-supplied initialization routine are passed in registers and in memory at execution time, and in external references at linkage edit time. The register contents expected on entry to the supplied initialization routine are listed in Table 9-15. The contents of registers not specified in Table 9-15 are indeterminate.

TABLE 9-15. Register Contents on Entry to Motorola-Supplied Initialization Routine

REGISTER	CONTENTS
D0.L	Loading task name
D1.L	Session number
D2.L	Default user volume
D3.W	Default user number
D4.L	First 4 characters of default catalog
D5.L	Second 4 characters of default catalog
D6.W	Command line length in bytes
D7.L	Bit mask of logical units passed
A1.L	Default terminal ID

The contents of registers D0, D2, D4, D5, and A1 are ASCII characters. D1, D3, and D6 contain binary numbers of the length specified in Table 9-15. Register D7 contains a bit mask indicating which logical units were passed by the operating system to the Pascal program. A bit set to 1 indicates that the corresponding logical unit was passed. The logical unit number is given by the position number of the bit, with bit 0 being the least significant bit and bit 31 the most significant. VERSAdos logical unit numbers are in the range 1 through 7. Bit 0 of the register may or may not be set, but logical unit 0 of VERSAdos is never passed and is never available for use.

Normally, VERSAdos will always pass logical units 5 and 6 to the Pascal program. Logical unit 5 is the VERSAdos command device/file logical unit number. When it is passed, the Motorola-supplied I/O routines make it the default assignment for the file identifier input. Logical unit 6 is the VERSAdos log device/file logical unit number. When it is passed, the Motorola-supplied I/O routines make it the default assignment for the file identifier output.

The parameters passed in memory are the command line and a word containing bits which select various runtime options. The command line is placed in memory at a fixed offset from an external label, .PZMAIN, defined in the Pascal program by Phase 2 of the compiler. At linkage edit time, it is assigned a fixed address of the first byte past the end of the RMA. Thus, .PZMAIN indicates to the initialization routine the address of the RMA.

The command line passed begins with the first non-space character of the parameter field of the command which invoked the Pascal program. Thus, the name of the program is not passed. The number of bytes passed in the command line is contained in register D6 on start-up.

The memory word specifying runtime options is at the external label, .PMAIN. Only one runtime option is defined. If bit 0 -- the least significant bit -- is set to 0, the initialization routine does not scan the command line for file assignments and options; if bit 0 is 1, then the initialization routine does scan the command line for file assignments and options. In both cases, the initialization routine scans the command line for the Z option, which permits the stack/heap area to be expanded during initialization.

The external label .PMAIN also specifies to the initialization routine the entry point of the Pascal program. Execution of the Pascal program begins at .PMAIN+2.

Several other external labels are used by the Motorola initialization routine. All of the external labels referenced by the initialization routine are listed in Table 9-16.

TABLE 9-16. Motorola-Supplied Initialization Routine External References

EXTERNAL LABEL	DESCRIPTION
.PMAIN	Runtime option bit mask and Pascal program entry point -2
.PZMAIN	Next location past RMA in section 15
.PZSTART	First location in section 15 allocated by the compiler
.PALSTS	Entry point to allocate stack storage subroutine
.PVTRAPD	Entry point to TRAP #13 handler
.PVTRAPE	Entry point to TRAP #14 handler
.PVBUSER	Entry point to bus error exception handler
.PADDRER	Entry point to address error exception handler
.PVZDIV	Entry point to divide by zero exception handler
.PVCHKI	Entry point to CHK instruction exception handler
.PVTRAPV	Entry point to TRAPV instruction exception handler
.POPTION	Entry point to option scanning

Initialization references .PALSTS to guarantee that the allocate stack space routine will always be loaded, even though initialization does not use it. The user-written initialization routine should also reference .PALSTS. .POPTION is the entry point of a subroutine which extracts options from the command line. Except for .PMAIN, .PZMAIN, .POPTION, .PZSTART, and .PALSTS, the remaining externals are entry points to the various trap and exception handlers Pascal programs require. The Motorola-supplied initialization must know these entry points so that it can request its own handling of these traps and exceptions from the operating system.

The entry point of the initialization routine .PINIT is specified as an operand in the end statement of the initialization routine. After VERSAdos loads the Pascal program, it starts execution of it at this location. Once the initialization routine has configured the runtime environment, it starts the Pascal program at .PMAIN+2.

The initialization routine must contain the long JSR routine and an external definition to it, .PLJSR.

An external reference to .PLJSR normally brings in the standard initialization module. LJSR is a routine which implements a long (greater than 32K-byte offset) branch to subroutine. Figure 9-6 contains the source of the LJSR routine, which should be included in the user initialization procedure. The compiler generates an external reference to .PLJSR, even though it might never use it, so that the initialization procedure will be included in the linkage edit.

```
*
* .PLJSR - LONG JSR ROUTINE
*
.PLJSR   EQU      *
         MOVE.L   (A7),A4              ! GET RETURN ADDRESS
         ADD.L    #4,(A7)              ! CORRECT IT ON STACK
         ADD.L    (A4),A4              ! CALCULATE DESTINATION
         JMP      (A4)                 ! GO TO IT
```

FIGURE 9-6. LJSR Routine

The standard initialization causes the loading of routines that handle certain traps and exceptions. These trap and exception handlers also rely on some VERSAdos functions. Thus, to be independent of VERSAdos, the user will have to instruct the linker to load replacement routines. Paragraph 9.8 describes the trap and exception handlers that must be replaced.

Most of the RMA is initialized by the initialization routine. The areas not initialized by it are: the statement counter field, the seven display levels, the unused areas, the command line, and the runtime error routine storage.

The command line is stored in the RMA in the VERSAdos environment by the task which loads and starts the Pascal program. The command (name of the Pascal program) and any spaces following it are not stored; only the parameters in the command line are passed in the command line field. The runtime error routine storage area is provided so that the error handler can save some of the processor's registers without the possibility of destroying the stack/heap contents.

The Pascal program initializes the statement counter field and the seven display level fields.

9.5.1 Initialization Sequence under VERSAdos on EXORmacs or VME/10

The first thing the Motorola-supplied initialization routine does is initialize A7. The value put in A7 is .PZMAIN minus the size of the RMA. This causes A7 to point to the first byte of the RMA. Next, the register contents passed on startup, as shown in Table 9-15, are saved in the appropriate areas of the RMA. A carriage return is put at the end of the command line. After this, the supplied initialization routine requests information about the memory segment containing the stack/heap area. It uses the RMA as a buffer area to receive the information. The main pieces of information are the size of the memory segment and the smallest address of the segment. Room is left at the beginning of the segment for the vector tables required by the executive so that the Pascal program can service some of the traps and exceptions. The address of the first location past the vector table is saved in the RMA as the smallest heap address, and is used as the beginning of the heap. The code in Figure 9-7 then initializes the beginning of the heap. This code places heap maintenance information in the heap. The address of the first location after the heap maintenance information is saved as the heap pointer. All user-supplied initialization routines should set up the heap in this manner. The largest address of the stack/heap segment is saved as the stack/heap segment break address in the RMA. This address is used by the initialize file descriptor routine to keep the file descriptor blocks from straddling a segment boundary. This could only occur if the stack/heap area would be increased by the Z option on the command line.

Next, the Motorola-supplied initialization routine clears (stores 0's in) the command line information area. This is the area of the RMA from just past the command line area up to the stack/heap segment break address. The vector tables at the beginning of the segment are initialized next and the vectors announced to the operating system. This lets the runtime routines service certain traps and exceptions.

The runtime option word at .PMAIN is checked next to determine if the command line is to be scanned for file assignments and options. If bit 0 of this location is 0, the command line is not scanned for file assignments. The command line is still scanned for the Z option but the command line information area of the RMA, just past the command line and up to the stack/heap segment break address, is not updated.

If bit 0 of the word at .PMAIN is 1, the command line is scanned for file assignments and options. The location in the RMA that contains the "# of fields in command line" is updated to contain the number of file name fields in the command line, not counting the I= and O= fields if they are present. The information for each file name field is also updated in the RMA. The number of files in each field is stored in the appropriate position of the RMA. This value is used to support file lists. The offset from the start of the command line, as saved in the RMA, to the start of the field is saved in the appropriate "offset to field" location in the RMA. The areas of the RMA labeled "current file in field" and "offset to current file" are used during Pascal program execution, and are not initialized beyond being cleared. If the initialization routine detects more than 16 file name fields, not counting the I= and O= fields, it reports an error and aborts the program. The options are also scanned and the offset to the option character from the start of the command line is saved in the array ['A'..'Z'] in the RMA. The offset is saved as a word value. Its position in the array is determined by the option letter.

For example, the first entry in the array is for the option character A, the second is for B, etc. If the option is preceded by a minus sign, the offset value saved in the array is the two's complement of the actual offset. This results in a negative entry in the array for minus options. The Z option is a special Pascal runtime option. Its presence is never indicated in the array even if it exists in the command line.

After the command line has been processed, the initialization routine determines if it needs to expand the stack-heap memory segment. If it found a Z option and the amount of stack/heap memory specified in the option is more than the current size, it attempts to add an additional memory segment at the top end (largest address) of the current segment. The size of the requested segment is equal to the size specified in the Z option minus the size of the current segment. Once the additional segment has been obtained, the RMA is moved up (to larger addresses) by an amount equal to the size of the added segment. Also, A7 is adjusted accordingly.

Next, the initialization routine sets up the register values needed for Pascal program execution. These register values are listed in Table 9-17. A3 points to the long JSR routine. A5, A6, and A7 all point to the first location of the RMA.

TABLE 9-17. Register Contents at End of Initialization

REGISTER	CONTENTS
A3.L	Points to long JSR routine
A5.L	Points to start of RMA
A6.L	Points to start of RMA
A7.L	Points to start of RMA

Then the initialization routine makes a call that will cause the "standard" floating point environment to be initialized, if required. The destination of the call is the first location of a section 8 common section named FINIT. This common section, as defined in the initialization routine, contains only an RTS instruction. However, if Phase 2 generated the appropriate external references to cause loading of the floating point routines, then they overwrite the common section with a call to standard floating point initialization, followed by an RTS instruction. The register contents of the MC68000/MC68010 are maintained across the call to the standard floating point initialization. No special initialization is required for fast floating point.

Finally, the initialization routine begins execution of the Pascal program. The entry point of the Pascal program is .PMAIN+2.

```
*
* A3 CONTAINS SMALLEST HEAP ADDRESS
*
        MOVE.L  #-1, (A3)+
        CLR.L   (A3)+
*
* A3 NOW CONTAINS CURRENT HEAP POINTER
* SAVE A3 IN RMA HEAP POINTER LOCATION
*
```

FIGURE 9-7. Heap Initialization

9.5.2 Initialization Sequence under VERSAdos on the VM01, VM02, and MVME110

Initialization begins with the allocation of the temporary data segment SEGT. This segment is used for all calculations until the address of the Pascal data segment SEG2 has been determined and until SEG2 has been expanded to its final size with the Z option. Note that the relative position of the Pascal code and data segments cannot be set at link time as in the EXORmacs or VME/10 case without requiring a large block of contiguous memory. This is because no MMU is present on the VM01, VM02, or MVME110 to translate arbitrary physical addresses into logical addresses which would reflect the correct relative positioning established at link time. Also, SEG2 cannot be expanded by adding another contiguous data segment as with the EXORmacs or VME/10, since no MMU is present to guarantee contiguous logical addresses in all cases. Instead, the original SEG2 must be deleted and a new, larger SEG2 can then be allocated.

Once SEGT has been allocated, A7 is set to point to the end of SEGT minus the size of the RMA. Thus, A7 points to the first byte of the RMA. Next, the register contents passed on startup, as shown in Table 9-15, are saved in the appropriate areas of the RMA. A carriage return is then put at the end of the command line. Next, space is allotted at the beginning of the segment for the vector tables required by the executive so that the Pascal program can service some of the traps and exceptions. The address of the first location past the vector table is saved in the RMA as the smallest heap address and is used as the beginning of the heap. Heap maintenance information is then placed in the heap (as shown in Figure 9-7), and the heap pointer is set to the address of the first location after the heap maintenance information. Note that all user-supplied initialization routines should set up the heap in this manner.

Next, the Motorola-supplied initialization routine clears (stores 0's in) the command line information area. This is the area of the RMA from just past the command line area up to the stack/heap segment break address. The vector tables at the beginning of the segment are then initialized and the vectors announced to the operating system. This allows the runtime routines to service certain traps and exceptions.

At this point, segment information about the Pascal data segment SEG2 is retrieved using an RMS68K directive. If SEG2 does not exist, an error is issued and the program is aborted. Otherwise, the information about SEG2 is used to calculate the address of the command line in SEG2. The command line is then copied into SEGT where it can be examined later.

The runtime option word at .PMAIN is checked next to determine if the command line is to be scanned for file assignments and options. If bit 0 of this location is 0, the command line is not scanned for file assignments. The command line is still scanned for the Z option, but the command line information area of the RMA, just past the command line and up to the stack/heap segment break address, is not updated.

If bit 0 of the word at .PMAIN is 1, the command line is scanned for file assignments and options. The location in the RMA that contains the "# of fields in the command line" is updated to contain the number of file name fields in the command line, not counting the I= and O= fields if they are present. The information for each file name field is also updated in the RMA. The number of files in each field is stored away in the RMA in order to support file lists later. The offset from the start of the command line to the start of the field, as saved in the RMA, is saved in the appropriate "offset to field" location in the RMA. The areas of the RMA labeled "current file in field" and "offset to current file" are used during Pascal program execution, and are not initialized beyond being cleared. If the initialization routine detects more than 16 file name fields, excluding the I= and O= fields, it reports an error and aborts the program. The options are also scanned, and the offset to the option character from the start of the command line is saved in the array ['A'..'Z'] in the RMA. The offset is saved as a word value, and its position in the array is determined by the option letter.

For example, the first entry in the array is for the option character A, the second is for B, etc. If the option is preceded by a minus sign, the offset value saved in the array is the two's complement of the actual offset. This results in a negative entry in the array for minus options. The Z option is a special Pascal runtime option. Its presence is never indicated in the array even if it exists in the command line.

After the command line has been processed, the initialization routine determines if it needs to expand the stack-heap memory segment. If it finds a Z option and the amount of stack/heap memory specified in the option is more than the current size, the original copy of SEG2 is deleted and SEG2 is reallocated with the larger segment size.

Once we have processed the Z option, there is no longer any need for the temporary segment SEGT. Thus, we now copy the RMA from SEGT into SEG2, initialize A7 to point into SEG2 instead of SEGT, reinitialize the heap information as before, and reinitialize the trap vectors as before. Finally, SEGT is deleted.

The initialization routine now proceeds to set up the register values needed for Pascal program execution. These register values are listed in Table 9-17. A3 points to the long JSR routine, while A5, A6, and A7 all point to the first location of the RMA.

Next the initialization routine makes a call that will cause the "standard" floating point environment to be initialized, if required. The destination of the call is the first location of a section 8 common section named FINIT. This common section, as defined in the initialization routine, contains only an RTS instruction. However, if Phase 2 generated the appropriate external references

to cause loading of the floating point routines, then they overwrite the common section with a call to standard floating point initialization, followed by an RTS instruction. The register contents of the MC68000/MC68010 are maintained across the call to the standard floating point initialization. No special initialization is required for fast floating point.

Finally, the initialization routine begins execution of the Pascal program. The entry point of the Pascal program is .PMAIN+2.

9.6 INITIALIZATION UNDER RMS68K ON VM01, VM02, OR MVME110

The Motorola-supplied initialization routine accomplishes several functions, including:

a. Setting up the RMA.
b. Requesting trap and exception vectors.
c. Setting up processor registers for the Pascal program.
d. Calling the floating point initialization routine, if required.

Depending on what the user has done with the other I/O routines and the configuration of the system, some of the initialization functions can be simplified or even eliminated. For example, if the user's I/O routines do not require all of the RMA field, the initialization of the RMA can be simplified. Most of the other initialization functions, except processor register initialization for the Pascal program, can be similarly reduced or eliminated, as permitted by the program's environment and I/O routines.

No parameters are required by the Motorola-supplied initialization routine under RMS68K. Instead, register values, which are normally supplied on startup under VERSAdos, are set to benign values for the more restricted RMS68K.

The contents of registers D0, D2, D4, D5, and A1 are ASCII characters. D1, D3, and D6 contain binary numbers of the length specified in Table 9-15. Register D7 contains the value zero, indicating that no logical units were passed to the Pascal program on startup.

Initialization references .PALSTS to guarantee that the allocate stack space routine will always be loaded, even though initialization does not use it. The user-written initialization routine should also reference .PALSTS. .POPTION is the entry point of a subroutine which extracts options from the command line. Except for .PMAIN, .PZMAIN, .POPTION, .PZSTART, and .PALSTS, the remaining externals are entry points to the various trap and exception handlers Pascal programs require. The Motorola-supplied initialization must know these entry points so that it can request its own handling of these traps and exceptions from the operating system.

Several other external labels are used by the Motorola initialization routine. All of the external labels referenced by the initialization routine are listed in Table 9-16.

The entry point of the initialization routine .PINIT is specified as an operand in the end statement of the initialization routine. After the sysgened system is booted up, the Pascal program contained in it starts execution at this location. After the initialization routine has configured the runtime environment, the user's Pascal program code begins execution at .PMAIN+2.

The initialization routine must contain the long JSR routine and an external definition to it, .PLJSR.

An external reference to .PLJSR normally brings in the standard initialization module. LJSR is a routine which implements a long (greater than 32K-byte offset) branch to subroutine. Figure 9-6 contains the source of the LJSR routine, which should be included in the user initialization procedure. The compiler generates an external reference to .PLJSR, even though it might never use it, so that the initialization procedure will be included in the linkage edit.

The standard initialization causes the loading of routines that handle certain traps and exceptions. These trap and exception handlers also rely on some RMS68K functions. Thus, to be independent of RMS68K, the user will have to instruct the linker to load replacement routines. Paragraph 9.8 describes the trap and exception handlers that must be replaced.

Most of the RMA is initialized by the initialization routine. The areas not initialized by it are: the seven display levels, the unused areas, the unused command line, and the runtime error routine storage.

The runtime error routine storage area is provided so that the error handler can save some of the processor's registers without the possibility of destroying the stack/heap contents.

The Pascal program initializes the statement counter field and the seven display level fields.

9.6.1 Initialization Sequence under RMS68K on the VM01, VM02, and MVME110

Initialization begins with the allocation of the temporary data segment SEGT. This segment is used for all calculations until the address of the Pascal data segment SEG2 has been determined. Note that the relative position of the Pascal code and data segments cannot be set at link time as in the EXORmacs or VME/10 case without requiring a large block of contiguous memory. This is because no MMU is present on the VM01, VM02, or MVME110 to translate arbitrary physical addresses into logical addresses which would reflect the correct relative positioning established at link time.

Once SEGT has been allocated, A7 is set to point to the end of SEGT minus the size of the RMA. Thus, A7 points to the first byte of the RMA. Next, the register contents which would normally be passed on startup by VERSAdos (as listed in Table 9-15), must be set to safe values for the RMS68K environment. Of particular interest are these:

a. D6.W is assumed to be 0 to indicate no command line is present.

b. D7.L is assumed to be 0 to indicate that no logical units are being passed on startup.

c. A1.L is assumed to be 'CN00' to indicate that the default terminal is the first serial port on the processor card.

Space is then allotted at the beginning of the segment for the vector tables required by the executive so that the Pascal program can service some of the traps and exceptions. The address of the first location past the vector table is saved in the RMA as the smallest heap address and is used as the beginning of the heap. Heap maintenance information is then placed in the heap, and the heap pointer is set to the address of the first location after the heap maintenance information. Note that all user-supplied initialization routines should set up the heap in this manner.

Next, the Motorola-supplied initialization routine clears (stores 0's in) the command line information area. This is the area of the RMA from just past the command line area up to the stack/heap segment break address. The vector tables at the beginning of the segment are then initialized and the vectors announced to the operating system. This allows the runtime routines to service certain traps and exceptions.

At this point, segment information about the Pascal data segment SEG2 is retrieved using an RMS68K directive. If SEG2 does not exist, an error is issued and the program is aborted. Otherwise, we can now use SEG2 in place of the temporary segment SEGT. Thus, we now copy the RMA from SEGT into SEG2, initialize A7 to point into SEG2 instead of SEGT, reinitialize the heap information as before, and reinitialize the trap vectors as before. Finally, SEGT is deleted.

The initialization routine now sets up the register values needed for Pascal program execution. These register values are listed in Table 9-17. A3 points to the long JSR routine, while A5, A6, and A7 all point to the first location of the RMA.

Next, the initialization routine makes a call that will cause the "standard" floating point environment to be initialized, if required. The destination of the call is the first location of a section 8 common section named FINIT. This common section, as defined in the initialization routine, contains only an RTS instruction. However, if Phase 2 generated the appropriate external references to cause loading of the floating point routines, then they overwrite the common section with a call to standard floating point initialization, followed by an RTS instruction. The register contents of the MC68000/MC68010 are maintained across the call to the standard floating point initialization. No special initialization is required for fast floating point.

Finally, the initialization routine begins execution of the Pascal program. The entry point of the Pascal program is .PMAIN+2.

9.7 THE RUNTIME INTERFACE TO BIOS UNDER RMS68K

9.7.1 Introduction

Pascal tasks have the capability of doing I/O under RMS68K, using the ports provided on the processor card of the VM01, VM02, or MVME110. To do this, users must SYSGEN BIOS with their Pascal task. To provide this support, the Pascal RTL provides two major functions. First, the routine CALCLU, given a device name, calculates the logical unit number that BIOS expects for the device. Users who wish to modify BIOS to support other devices need only modify CALCLU to translate the new device into the appropriate logical unit number. See paragraph 9.7.2 for a detailed description of CALCLU. The second function of the Pascal RTL is to establish a default terminal ID which will be assigned to the default Pascal text files INPUT and OUTPUT. This is done in the INIT procedure as described in paragraph 9.6.1. For specialized applications, users may want to change the default value of 'CN00'.

9.7.2 CALCLU Routine for RMS68K

CALCLU is a Pascal runtime routine that calculates the logical unit number (LUN) associated with a given device. As written, CALCLU recognizes four device names:

```
#PR       (for a printer)
#CN01     (for a second terminal)
#CN00     (for a standard terminal)
#         (for a standard terminal)
```

For #PR, CALCLU calculates an LUN of 3; for #CN01, it calculates an LUN of 4; for #CN00 or #, an LUN of 5 is calculated for input, or an LUN of 6 for output.

Whenever a device driver is added to BIOS, a SETLU macro call must be included in the CALCLU routine of BIOS. The parameters of this macro call are device name and LU, in the format:

```
SETLU '<device name>',<logical unit number>
```

The call should be included after the conditional statement IFEQ RMSVM with the other SETLU calls.

9.8 TRAP VECTORS

Motorola Pascal running under VERSAdos requires that certain trap vectors be under control of the Pascal program. These vectors are used to aid in debugging programs and are not necessary in a debugged program. There are seven vectors used by Pascal programs. The seven used are:

a. TRAP #13 instruction
b. TRAP #14 instruction
c. Trap on Overflow (TRAPV) instruction
d. Check (CHK) instruction
e. Divide by zero
f. Address Error
g. Bus Error

TRAP #13 is used to indicate failure during range checking of some item which cannot be checked using the MC68000/MC68010 CHK instruction.

TRAP #14 is a general purpose error exit and may be invoked by either the Pascal program code or the runtime library routines. TRAP #14's are followed by one word in memory describing what kind of an error occurred. There are currently only five types of TRAP #14:

a. A 0 in the word after the trap means an error has occurred in a Pascal runtime routine, and the error number is in D0. Only the low order byte of D0 is significant.

b. A 1 in the word after the trap means it was a floating point runtime error. The floating point error number is in the low order word of D0.

c. A 2 in the word after the trap means it was an error returned by either the FHS or IOS tasks. The relevant FHS/IOS error is in the lowest order byte of the D0 register.

d. A 3 in the word after the trap instruction means that a runtime error has occurred while executing in-line code -- such as a case index out of range error. The error number in this case is one word on the stack under the information pushed by the trap call.

e. A 4 in the word after the trap instruction means the standard procedure HALT(n) was used in the Pascal source program. A 2-byte integer, n, is on the stack under the information pushed by the trap call. NOTE: HALT(0) means perform a normal termination; otherwise, an abort with the user's error number is done. HALT will not cause buffers associated with open files to be flushed.

Trap on overflow is currently unused, but is reserved for future enhancements.

The check instruction vector is required for range check of some items of 16 bits or less.

The divide-by-zero vector is required to catch divide-by-zero errors and report them properly.

The bus error vector is used to help in determining where and how a program is trying to access memory outside its range.

The address error vector is used to help in detecting stack/heap overflow and compiler mistakes.

If control is given to a routine via a trap vector, other than TRAP #14 with a word containing 4 following and a word of 0 on the stack below the trap information, it means that the program is terminating abnormally.

9.9 RUNTIME INTERFACE FOR VERSAmodule 01 Under VERSAbug

The Pascal runtime routines for the VERSAmodule 01 assume that VERSAbug is residing in ROM, and is configured as described in the "VERSAbug Operating Procedure" section of the VERSAbug Debugging Package User's Guide. To use Pascal without VERSAbug, the DEBUG entry in the Control Table (paragraph 9.9.1) must be changed and the runtime routines described in the following paragraphs may be altered to suit the user's purpose.

The layout of the Runtime Maintenance Area (RMA) for VERSAmodule 01 is shown in Table 9-18. This table corresponds to the areas of the RMA for VERSAdos (Table 9-14) which can't be redefined.

TABLE 9-18. VERSAmodule 01 Runtime Maintenance Area

Offsets from A5		
hex	decimal	
0	0	Statement Counter
4	4	Heap Pointer
8	8	Display Level 0
C	12	Display Level 1
10	16	Display Level 2
14	20	Display Level 3
18	24	Display Level 4
1C	28	Display Level 5
20	32	Display Level 6
24	36	Display Level 7
28	40	Not Used $148 bytes (328 decimal)
170	368	Smallest Heap Pointer Value
174	372	Runtime Error Routine Storage $48 bytes (72 decimal)
1BC	444	

The VERSAmodule 01 file parameter block is shown in Table 9-19. The Device ID is a 4-character device name, and the DDT Address is a 4-byte address of the Device Descriptor Table. The other field descriptions are as for the previous description of the file parameter block for Pascal under VERSAdos.

TABLE 9-19. VERSAmodule 01 File Parameter Block

Offsets hex	Offsets decimal	
0	0	File Position
4	4	Pascal File Status | Component Size
8	8	Device ID
C	12	DDT Address
10	16	Number of bytes input
14	20	Address of buffer start
18	24	Address of buffer end
1C	28	Record buffer (134 bytes if text file; component size if not)

All the runtime routines are position independent, except for the Device Service Address (DSA) entries in the Device Descriptor Table (DDT) (paragraph 9.9.2) and the stack/heap start and end addresses -- MEMBEG and MEMEND -- in the Control Table (paragraph 9.9.1). These addresses are all the addresses that define where the RAM for the Pascal program resides. Thus, the program can be moved, but its RAM will remain at a fixed address.

The routine .PINI, in the module INIT, sets the trap vectors. It assumes that the trap vectors are stored in RAM. If the trap vectors are to be put in ROM, the routine INIT must be edited so that the line MEM$TYP EQU RAM changes to MEM$TYP EQU ROM. Then assemble the runtime routines and reconstruct the library.

9.9.1 Control Table

The Control Table describes general control information. Its format is as follows:

```
DDTLINK  DC.L    PRT1DDT-*      DDT link
.PVTRAP1 BRA     TRP1HNDL       Trap 1 handler
CBREAKB  BRA     CBREAK         Check for break
CBREAKAD DC.L    SER_IO1        Break serial I/O address
CBREAKHB BRA     CBREAKH        Break handler
CWAITB   BRA     CWAIT          Check for wait
CWAITAD  DC.L    SER_IO1        Wait serial I/O address
CWAITCH  DC.B    $17            Wait character (CTRL-W)
         DC.B    0              Not used
EOF      DC.L    1              End of file error
START    BRA     .PSTART        Routine to start execution
STD$DV   DC.B    'PRT1'         INPUT/OUTPUT files default to PRT1;
OTH$DV   DC.B    'PR  '          all other files default to PRT2.
DEBUG    DC.L    $F00030        Address of VERSAbug pause routine,
*                                or 0 if no VERSAbug present.
MEMBEG   DC.L    FIRST          Lowest address of stack/heap
MEMEND   DC.L    .PZMAIN        Highest address of stack/heap
         END     START
```

where:

DDTLINK — is the offset from the start of the Control Table to the start of the Device Descriptor Table (DDT).

.PVTRAP1 — is a branch to the TRAP #1 handler. A TRAP #1 is taken to terminate the program. Both normal and error terminations are handled by this routine. The D0 register contains $0F if there was a normal termination; otherwise, it contains $0E for any other termination. A0 contains the error code if D0 contains $0E. This routine will stop the program and go to VERSAbug if the DEBUG flag is set (non-zero). If the flag is not set (zero), then VERSAbug is not resident and the program will execute a STOP instruction.

CBREAKB — is a branch to the routine to check for a break. The I/O system calls this before reading or writing a character. This routine checks to see if the user has pressed the BREAK key. If pressed, the system calls the CBREAKH routine to service the break.

CBREAKAD — is the address of the serial port to check the status of the BREAK key. The default value of this entry is serial port 1, which is usually connected to a user's terminal.

CBREAKHB — is a branch to the routine that will service the break if CBREAK detects one. If the DEBUG flag is set, it will return control to VERSAbug. A VERSAbug GO command will continue execution. If the flag is zero, then there is no VERSAbug and break is ignored.

CWAITB — is a branch to the routine to see if a CTRL-W has been typed on the device specified by CWAITAD. If it has, this routine will not return until something else is typed. This routine is called before each character is output. This allows the user to type a CTRL-W to suspend output in order to view it.

CWAITAD is the address of the serial port to check for CTRL-W.

CWAITCH is the character CTRL-W to wait on.

EOF is the status returned by the read routines when they sense an end of file condition. On the terminals, this means that the user typed a CTRL-Z.

START is the first location to be executed. It is a branch to a routine that figures out how much memory is available to the program, and does a BRA .PINIT with the A7 register containing the address of top of stack/heap, and the A1 register containing the address of bottom of stack/heap.

STD$DV is the name of the default device for the files INPUT and OUTPUT.

OTH$DV is the name of the default device for all other files.

DEBUG is the address where Pascal will go when the BREAK key is pressed or the program is terminated. If this address is 0, a break is ignored and program termination is done with a STOP #0 instruction. The default value of this location assumes that VERSAbug is in its usual location at $F00000.

MEMBEG contains the first location of the stack/heap. This defaults to the first word of memory after the DSA.

MEMEND contains the last location of the stack/heap. This defaults to the value of the symbol .PZMAIN, which is defined by the Pascal compiler.

9.9.2 Device Descriptor Table

The Device Descriptor Table (DDT) has one entry for each device in the system. Initially, VERSAmodule 01 is set up for three devices, defined as follows:

```
#PRT1     Serial port 1 (normally connected to a user's terminal)
#PRT2     Serial port 2 (normally connected to a host computer)
#PR       Parallel port 1, interfaced to a printer.
```

This selection can be changed by modifying the DDT.

An example format of an entry in the DDT is the following entry for serial port 1:

```
PRT1DDT  DC.L    PRT2DDT-*          #1.  Link
         DC.L    'PRT1'             #2.  Device name
         BRA     AINIT              #3.  Initialization subroutine
         BRA     ARST               #4.  Reset subroutine
         BRA     ARWT               #5.  Rewrite subroutine
         BRA     P1RD               #6.  Read subroutine
         BRA     AWRT               #7.  Write subroutine
         BRA     ACLO               #8.  Close subroutine
         DC.L    SER_IO1            #9.  Device address
         DC.L    P1DSA              #10. DSA address
*
* Initialization data (ASCII device data shown)
*
         DC.B    0                  #11. Character null pad count
         DC.B    0                  #12. RETURN null pad count
         DC.B    $1A                #13. End-of-file character (CTRL-Z)
         DC.B    $15                #14. Serial I/O control character
         DC.B    $7F                #15. Delete character (DEL)
         DC.B    $18                #16. Cancel line (CTRL-X)
         DC.B    $04                #17. Echo line (CTRL-D)
         DC.B    0                  #18. Not used
```

where:

#1 is the link; the offset to the next entry in the DDT, or 0 if this is the last entry.

#2 is the device name.

#3 is the call to the initialization subroutine.

#4-#8 are subroutine entries; called whenever a reset, rewrite, read, write, or close call is made to the device. The register formats upon entry and exit are described in Table 9-20.

#9 is the device address.

#10 is the DSA address; the address (in RAM) of any variables used by this device. This is provided so that a single subroutine can service several devices. The device initialization should copy the initial data from the DDT entry into this area.

TABLE 9-20. Register Formats for I/O Subroutines (VERSAmodule 01)

SUBROUTINE	TIME	REGISTER	CONTENTS
Device initialization subroutine	on entry	D7.L	Return address from initialization function.
		A0.L	Pointer to current DDT.
		A3.L	Address of long JSR routine.
		A5.L	Global variable pointer.
		A6.L	Local variable pointer.
		A7.L	Stack pointer.
	on exit	D7,A0,A3,A5,A6	Are preserved
Reset subroutine	on entry	D1.L	Pascal file status.
		D7.L	Return address to Pascal program.
		A0.L	Pointer to current DDT.
		A2.L	Pointer to file descriptor.
		A3.L	Address of long JSR routine.
		A5.L	Global variable pointer.
		A6.L	Local variable pointer.
		A7.L	Stack pointer.
	on exit	D0.L	Error status.
		D1,D7,A2-A7	Are preserved.
		CCR.Z	Set if no error; zero otherwise.
Rewrite subroutine	on entry	D1.L	Pascal file status.
		D7.L	Return address to Pascal program.
		A0.L	Pointer to current DDT.
		A2.L	Pointer to file descriptor.
		A3.L	Address of long JSR routine.
		A5.L	Global variable pointer.
		A6.L	Local variable pointer.
		A7.L	Stack pointer.
	on exit	D0.L	Error status.
		D1,D7,A0,A2-A7	Are preserved.
		CCR.Z	Set if no error; zero otherwise.

TABLE 9-20. Register Formats for I/O Subroutines (VERSAmodule 01) (cont'd)

SUBROUTINE	TIME	REGISTER	CONTENTS
Read Subroutine	on entry	D1.L	Pascal file status.
		A0.L	Pointer to current DDT.
		A2.L	Pointer to file descriptor.
		A3.L	Address of long JSR routine.
		A5.L	Global variable pointer.
		A6.L	Local variable pointer.
		A7.L	Stack pointer.
	on exit	D0.L	Error status.
		D1-D7,A0-A7	Are preserved.
		CCR.Z	Set if no error; zero otherwise.
Write Subroutine	on entry	D1.L	Pascal file status.
		A0.L	Pointer to current DDT.
		A2.L	Pointer to file descriptor.
		A3.L	Address of long JSR routine.
		A5.L	Global variable pointer.
		A6.L	Local variable pointer.
		A7.L	Stack pointer.
	on exit	D0.L	Error status.
		D1-D7,A0-A7	Are preserved.
		CCR.Z	Set if no error; zero otherwise.
Close Subroutine	on entry	D1.L	Pascal file status.
		D7.L	Return address to Pascal program.
		A0.L	Pointer to current DDT.
		A2.L	Pointer to file descriptor.
		A3.L	Address of long JSR routine.
		A5.L	Global variable pointer.
		A6.L	Local variable pointer.
		A7.L	Stack pointer.
	on exit	D0.L	Error status.
		D1,D7,A0,A2-A7	Are preserved.
		CCR.Z	Set if no error; zero otherwise.

The initialization data is the data that will be copied into the DSA by the device initialization routine. For ASCII devices, the data is formatted as follows:

#11 is the character null pad count; the number of nulls to put after a character.

#12 is the carriage return null pad count; the number of nulls to put after a RETURN.

#13 is the end-of-file character (CTRL-Z); the character that tells the read routine to set the EOF status in the file variable.

#14 is the serial I/O control character; not used by any of the I/O subroutines since they assume that the serial I/O ports are already set up. This byte would normally contain the initial control word for the serial ports.

#15-#17 are the delete character (DEL), cancel line (CTRL-X), and echo line (CTRL-D) characters; used for input editing when reading from the terminal.

#18 is not used; defined so that the DDT entry will take up an even number of bytes.

9.9.3 Standard I/O Routines

Four standard I/O routines are provided:

P1RD If the file is of type TEXT, this routine will read a line from the terminal accepting the cancel, echo, and delete editing characters. If the file is of any other type, it will read unformatted until it reads enough bytes to fill the buffer.

P2RD This routine is the same as P1RD, except it will not allow editing characters.

AWRT This routine writes out a line to a serial port. The line is followed by a CR/LF if the file is of type TEXT. If the file is binary, it will send out the component one byte at a time without formatting.

PWRT This routine is the same as AWRT, except it writes out a line to a parallel port.

9.9.4 Initialization

The VERSAmodule initialization routine performs the same functions as the EXORmacs and VME/10 initialization routine (paragraph 9.5), but is much simpler due to the elimination of much of the RMA initialization, command line scanning, and VERSAdos interfacing. The VERSAmodule initialization routine accomplishes several functions, including:

a) Setting up the RMA.
b) Initializing the trap vectors.
c) Setting up processor registers for the Pascal program.
d) Initializing the devices.

Note that a user-written initialization routine for VERSAmodule must also contain the code shown in Figures 9-6 and 9-7. In addition, the initialization routine expects register A1 to contain the address of the bottom of the stack/heap (see MEMBEG in the control table) and register A7 to contain the address of the top of the stack/heap (see MEMEND in the control table).

9.9.5 Table Listing

```
MOTOROLA M68000 ASM      FIX :2560.        .TABLE   .SA

PASCAL I/O CONTROL TABLES

   2                        TABLE    IDNT     0,0                  IO DEFINITIONS TABLES
   3                        ********************************************
   4                        *
   5                        *
   6                        *
   7                        *       P A S C A L   I/O   ROUTINES
   8                        *
   9                        *      COPYRIGHTED 1980 BY MOTOROLA, INC.
  10                        *
  11                        *
  12                        *
  13                        ********************************************
  14                        *
  15                        * CONTROL TABLE
  16                        * CONTAINS CONTROL BRANCHES AND VARIABLES
  17                        *
  18             00000008            SECTION  8
  19                                 XREF     TRP1HNDL,CBREAK,CBREAKH,CWAIT
  20                                 XDEF     .PTABLE,DDTLINK,CBREAKB,CBREAKAD,CBREAKHB
  21                                 XDEF     CWAITB,CWAITAD,CWAITCH
  22                                 XDEF     STD$DV,OTH$DV,START
  23                                 XDEF     DEBUG
  24                                 XDEF     MEMBEG,MEMEND
  25                                 XREF     .PZMAIN
  26                                 XREF     .PSTART
  27                                 XDEF     EOF
  28                                 XDEF     .PVTRAP1
  29 8           00000000  .PTABLE  EQU      *
  30 8 00000000  0000003A  DDTLINK  DC.L     PRT1DDT-*            DDT LINK
  31 8 00000004  6000FFFA  .PVTRAP1 BRA      TRP1HNDL             TRAP 1 HANDLER
  32 8 00000008  6000FFF6  CBREAKB  BRA      CBREAK               CHECK FOR BREAK
  33 8 0000000C  00F70011  CBREAKAD DC.L     SER_IO1              BREAK SER_IO ADDR
  34 8 00000010  6000FFEE  CBREAKHB BRA      CBREAKH              BREAK HANDLER
  35 8 00000014  6000FFEA  CWAITB   BRA      CWAIT                CHECK FOR WAIT
  36 8 00000018  00F70011  CWAITAD  DC.L     SER_IO1              WAIT SER_IO ADDR
  37 8 0000001C  17        CWAITCH  DC.B     $17                  WAIT CHAR (CTL-W)
  38 8 0000001D  00                 DC.B     0                    NOT USED
  39 8 0000001E  00000001  EOF      DC.L     1                    END OF FILE ERROR (DREAD)
  40 8 00000022  6000FFDC  START    BRA      .PSTART              START ROUTINE
  41 8 00000026  50525431  STD$DV   DC.B     'PRT1'               INPUT/OUTPUT DEFAULT DEVICE
  42 8 0000002A  50522020  OTH$DV   DC.B     'PR  '               DEFAULT FOR OTHER FILES
```

```
 43 8 0000002E 00F00030     DEBUG    DC.L     $F00030              ADDRESS OF DEBUG
 44 8 00000032 00000018     MEMBEG   DC.L     FIRST
 45 8 00000036 00000000     MEMEND   DC.L     .PZMAIN
 46                         *
 47                         *
 48                         * VERSAMOULE EQUATES
 49                         *
 50            00F70011     SER_IO1  EQU      $F70011              TERMINAL SER_IO
 51            00F70019     SER_IO2  EQU      $F70019              HOST SER_IO
 52            00F70020     PDIT     EQU      $F70020
 53
 54                         *
 55                         * LABELS TO DSA ENTRIES
 56                         *
 57                                  OFFSET   0
 58   00000000 00000001     CHRNL    DS.B     1                    CHAR NULL PAD COUNT
 59   00000001 00000001     CRNL     DS.B     1                    CR NULL PAD COUNT
 60   00000002 00000001     EOFCHR   DS.B     1                    EOF CHAR
 61   00000003 00000001              DS.B     1                    SER_IO CONTROL BYTE
 62   00000004 00000001     DELCHR   DS.B     1                    DELETE CHAR CHAR
 63   00000005 00000001     CANCHR   DS.B     1                    CANCEL LINE CHAR
 64   00000006 00000001     ECHOCHR  DS.B     1                    ECHO LINE CHAR
 65
 66                         *
 67                         * VERSAMODULE DEVICE DESCRIPTOR TABLES (DDT)
 68                         *
 69            00000008              SECTION  8
 70                                  XREF     AINIT,ARST,ARWT,P1RD,AWRT,ACLO
 71 8 0000003A 00000030     PRT1DDT  DC.L     PRT2DDT-*            LINK
 72 8 0000003E 50525431              DC.L     'PRT1'               DEVICE ID
 73 8 00000042 6000FFBC              BRA      AINIT                INIT SUB
 74 8 00000046 6000FFB8              BRA      ARST                 RST SUB
 75 8 0000004A 6000FFB4              BRA      ARWT                 RWT SUB
 76 8 0000004E 6000FFB0              BRA      P1RD                 READ SUB
 77 8 00000052 6000FFAC              BRA      AWRT                 WRITE SUB
 78 8 00000056 6000FFA8              BRA      ACLO                 CLO SUB
 79 8 0000005A 00F70011              DC.L     SER_IO1              DEV ADDR
 80 8 0000005E 00000000              DC.L     P1DSA                DSA ADDR
 81 8 00000062 00                    DC.B     0                    CHAR NULL PAD COUNT INIT
 82 8 00000063 00                    DC.B     0                    CR NULL PAD COUNT INIT
 83 8 00000064 1A                    DC.B     $1A                  EOF CHAR (CTL-Z) INIT
 84 8 00000065 15                    DC.B     $15                  SER_IO CTL CHAR INIT
 85 8 00000066 7F                    DC.B     $7F                  DELETE CHAR (DEL)
 86 8 00000067 18                    DC.B     $18                  CANCEL LINE (CTL-X)
 87 8 00000068 04                    DC.B     $04                  ECHO LINE (CTL-D)
 88 8 00000069 00                    DC.B     0                    NOT USED
 89
 90                                  XREF     P2RD
 91 8 0000006A 00000030     PRT2DDT  DC.L     PRDD-*               LINK
 92 8 0000006E 50525432              DC.L     'PRT2'               DEVICE ID
 93 8 00000072 6000FF8C              BRA      AINIT                INIT SUB
 94 8 00000076 6000FF88              BRA      ARST                 RST SUB
 95 8 0000007A 6000FF84              BRA      ARWT                 RWT SUB
 96 8 0000007E 6000FF80              BRA      P2RD                 READ SUB
 97 8 00000082 6000FF7C              BRA      AWRT                 WRITE SUB
 98 8 00000086 6000FF78              BRA      ACLO                 CLO SUB
 99 8 0000008A 00F70019              DC.L     SER_IO2              DEV ADDR
100 8 0000008E 00000008              DC.L     P2DSA                DSA ADDR
101 8 00000092 00                    DC.B     0                    CHAR NULL PAD COUNT INIT
102 8 00000093 00                    DC.B     0                    CR NULL PAD COUNT INIT
103 8 00000094 1A                    DC.B     $1A                  EOF CHAR (CTL-Z) INIT
104 8 00000095 15                    DC.B     $15                  SER_IO CTL CHAR INIT
105 8 00000096 7F                    DC.B     $7F                  DELETE CHAR (DEL)
106 8 00000097 18                    DC.B     $18                  CANCEL LINE (CTL-X)
107 8 00000098 04                    DC.B     $04                  ECHO LINE (CTL-D)
108 8 00000099 00                    DC.B     0                    NOT USED
109                                  XREF     ILL,PWRT
110 8 0000009A 00000000     PRDD     DC.L     0                    LINK
111 8 0000009E 50522020              DC.L     'PR  '               DEVICE ID
112 8 000000A2 6000FF5C              BRA      AINIT                INIT SUB
113 8 000000A6 6000FF58              BRA      ILL                  RST SUB
114 8 000000AA 6000FF54              BRA      ARWT                 RWT SUB
```

```
 115 8 000000AE 6000FF50              BRA      ILL                  READ SUB
 116 8 000000B2 6000FF4C              BRA      PWRT                 WRITE SUB
 117 8 000000B6 6000FF48              BRA      ACLO                 CLO SUB
 118 8 000000BA 00F70020              DC.L     PDI1                 DEV ADDR
 119 8 000000BE 00000010              DC.L     PRDSA                DSA ADDR
 120 8 000000C2 00                    DC.B     0                    CHAR NULL PAD COUNT INIT
 121 8 000000C3 00                    DC.B     0                    CR NULL PAD COUNT INIT
 122 8 000000C4 1A                    DC.B     $1A                  EOF CHAR (CTL-Z) INIT
 123 8 000000C5 15                    DC.B     $15                  SER IO CTL CHAR INIT
 124 8 000000C6 7F                    DC.B     $7F                  DELETE CHAR (DEL)
 125 8 000000C7 18                    DC.B     $18                  CANCEL LINE (CTL-X)
 126 8 000000C8 04                    DC.B     $04                  ECHO LINE (CTL-D)
 127 8 000000C9 00                    DC.B     0                    NOT USED
 128            00000000              SECTION  0
 129 0 00000000 00000008     P1DSA    DS.B     8
 130 0 00000008 00000008     P2DSA    DS.B     8
 131 0 00000010 00000008     PRDSA    DS.B     8
 132 0 00000018 00000000     FIRST    DC.L     0                    FIRST ADDRESS OF STACK-HEAP
 133                         * SECTION 15 MUST FOLLOW THIS ADDRESS!!!!!!!!!
 134 8          00000022              END      START

****** TOTAL ERRORS      0--
****** TOTAL WARNINGS    0--


SYMBOL TABLE LISTING

SYMBOL NAME     SECT   VALUE       SYMBOL NAME     SECT   VALUE


.PSTART     XREF  *  00000000     ECHOCHR                00000006
.PTABLE     XDEF  8  00000000     EOF         XDEF  8    0000001E
.PVTRAP1    XDEF  8  00000004     EOFCHR                 00000002
.PZMAIN     XREF  *  00000000     FIRST                  00000018
ACLO        XREF  *  00000000     ILL         XREF  *    00000000
AINIT       XREF  *  00000000     MEMBEG      XDEF  8    00000032
ARST        XREF  *  00000000     MEMEND      XDEF  8    00000036
ARWT        XREF  *  00000000     OTH$DV      XDEF  8    0000002A
AWRT        XREF  *  00000000     P1DSA                  00000000
CANCHR               00000005     P1RD        XREF  *    00000000
CBREAK      XREF  *  00000000     P2DSA                  00000008
CBREAKAD    XDEF  8  0000000C     P2RD        XREF  *    00000000
CBREAKB     XDEF  8  00000008     PDI1                   00F70020
CBREAKH     XREF  *  00000000     PRDD                8  0000009A
CBREAKHB    XDEF  8  00000010     PRDSA               0  00000010
CHRNL                00000000     PRT1DDT             8  0000003A
CRNL                 00000001     PRT2DDT             8  0000006A
CWAIT       XREF  *  00000000     PWRT        XREF  *    00000000
CWAITAD     XDEF  8  00000018     SER_IO1                00F70011
CWAITB      XDEF  8  00000014     SER_IO2                00F70019
CWAITCH     XDEF  8  0000001C     START       XDEF  8    00000022
DDTLINK     XDEF  8  00000000     STD$DV      XDEF  8    00000026
DEBUG       XDEF  8  0000002E     TRP1HNDL    XREF  *    00000000
DELCHR               00000004
```

CHAPTER 10

FLOATING POINT ROUTINES

10.1 IMPLEMENTATION

M68000 Family Pascal offers two versions of floating point software: the IEEE standard floating point processor simulator (M68341), which supports 32-, 64-, and 80-bit numbers; and the single-precision fast floating point software package (M68343), which supports 32-bit numbers only. The fast floating point (FFP) software, selectable with Pascal's Q option (Phase 1 command line or option comment in source program), offers a significant speed advantage over the default standard version.

The floating point routines are made available when the Pascal program is linked to Pascal's default runtime library, PASCALIB.RO. For fast floating point (FFP), all routines are separately-callable subroutines in PASCALIB. In standard floating point (standard FP), most real functions are performed by software simulation of a floating point processor; several functions are performed by separately-callable subroutines in PASCALIB.

Modules which use fast floating point may not be linked with modules which use the standard floating point. If this is attempted, the linker will display the error message:

```
** WARNING 701 - Multiply defined symbol:  .PFLOATP
```

This chapter describes the interface between M68000 Family Pascal and the floating point routines, and lists the real functions provided in the default runtime library; the user may wish to replace certain routines with his own 32-bit, IEEE-format-compatible routines.

Chapter 11 provides an internal representation of data which is applicable to either version of floating point, and a brief glossary of floating point terms. Appendix A provides a description of the standard FP processor.

10.1.1 Interface to Floating Point Processor (Standard FP)

Instructions to the standard floating point processor take the form of the M68000/M68010 reserved F-line instruction. Thus, when the M68000/M68010 encounters a floating point processor instruction, it generates a trap to the F-line exception handler. This handler then sets up the appropriate environment for the floating point processor, and executes it.

However, in the EXORmacs environment, the F-line trap would return control to the executive, which would determine the cause of the trap and finally give control back to the Pascal F-line trap handler. In order to bypass this overhead, a call to an F-line trap simulator is inserted before each F-line instruction.

The F-line trap simulator sets up the environment as if an F-line trap had occurred and the F-line trap handler had serviced it, and then calls the floating point processor.

The Phase 2 command line option -J inhibits the generation of the call to the F-line trap simulator. In this case, the F-line instructions generated would be executed. Thus, when using the -J option, the user must supply his own F-line trap handler, memory fetch/store routine, and floating point initialization routine. Appendix A describes the requirements of the F-line trap handler and the memory fetch/store routine.

10.1.2 Externals (Standard FP)

A number of external label references and definitions are used by the floating point software. If Phase 2 generates any floating point instructions, it will also generate an external reference to .PFINIT in order to bring in the floating point initialization routine. Additionally, if the Phase 2 command line option J is enabled and floating point instructions are generated, then .PFPOINT will be referenced as an external. .PFPOINT is the entry point of the F-line trap simulator. If the J option is disabled for the Phase 2 compilation of the main program (i.e., -J had been entered as a command option), then an RTS instruction will be generated in section 8 and defined as the external definition for .PFPOINT. This results in floating point instructions which exist in the runtime library to be executed as F-line instructions, even though they are preceded by a call to .PFPOINT.

The F-line trap simulator routine references .P68341P, which is the entry point to the floating point processor. This causes the linkage editor to include the floating point processor. The F-line trap simulator also includes an area in section 15 for the floating point register block required by the floating point processor. However, the block is not defined externally, since the F-line trap simulator passes the address of the block to the floating point processor as an entry parameter. Any user supplied F-line trap handler should supply its own floating point register block definition in section 15.

10.1.3 Floating Point Initialization (Standard FP)

The floating point initialization routine is called from the main initialization routine, as described in paragraph 9.5.1. First, the floating point initialization routine saves on the stack all the registers it will be using. Floating point registers occupy a space in memory directly above the RMA.

Next, it initializes a floating point exception vector table starting at the smallest heap pointer value as defined in the RMA (floating point exception vectors are located between the heap and the Pascal exception vectors). The structure of the table is shown in Figure 10-1. The starting address of the table gets passed to the floating point processor as an entry parameter. The default exception handler is included in the .PFINIT module. The exception handler loads the appropriate error number in A0 and then aborts the program.

The floating point initialization routine then updates the smallest heap pointer value in the RMA to point just after the vector table. The next step is to re-initialize the heap and heap pointer, as shown in Figure 9-8.

The floating point initialization routine then clears the STATUS and CNTRL registers of the floating point processor. Finally, it restores the registers it saved on entry and executes an RTS.

The supplied floating point initialization routine includes the instructions in the FINIT named common section that cause the floating point initialization to be called (see paragraph 9.5.1). Due to the order in which the linkage edit must be done, a user-supplied floating point initialization routine should not include the FINIT named common section. The user-supplied FINIT named common should be in a separate module and reference the beginning of the user's floating point initialization routine as an external. The linkage edit sequence that must be followed with user-supplied real runtime routines is:

a. Load the Pascal program modules.

b. Load the user-supplied runtime routines, including floating point processor initialization, F-line trap handler, etc.

c. Read the standard Pascal runtime library.

d. If using a user-supplied floating point processor initialization routine, load the user-supplied FINIT common section.

```
FPVEC  DC.L   INVALID.OPERATION
       DC.L   OVERFLOW
       DC.L   UNDERFLOW
       DC.L   DIVIDE.BY.ZERO
       DC.L   INEXACT.RESULT
       DC.L   INTEGER.OVERFLOW
       DC.L   RESERVED.EXPONENT
```

FIGURE 10-1. Floating Point Exception Vector Table

10.2 REAL RUNTIME ROUTINES (STANDARD FP)

This section describes real runtime routines that the user might want to replace. The descriptions include the external labels used to call the routine, the arguments and result of the routine, and the entry conditions that would cause a runtime error.

Each routine is written as a subroutine. All functions not listed here are handled by the floating point processor. The processor contains four floating point registers named FP0, FP1, FP2, and FP3 (all of type xreal), plus status and control registers. Registers A3, A5, and A6 must be preserved through the routine. Although they may be used by the routine, their original contents must be restored upon exit from the routine. Other registers may be modified by the routine.

The floating point processor is fully described in Appendix A.

The trap STATUS register of the floating point processor is set appropriately for any errors detected by the routines. If the trap is enabled, then the corresponding trap handler (as indicated in the floating point exception vector table) is called by the runtime routine. The trap stack environment is set up as if the trap handler had been called by the floating point processor.

10.2.1 Sine

External label: .PSIN

Entry:	FP0	Argument in radians.
	A7.L	Pointer to return address on the stack.
Exit:	FP0	Sine of entry FP0.
	A7.L	Pointer to the stack with the return address removed.

Description: Returns sine of value passed in FP0. Generates invalid operation error if entry value is a NaN or an infinity.

10.2.2 Cosine

External label: .PCOS

Entry:	FP0	Argument in radians.
	A7.L	Pointer to return address on the stack.
Exit:	FP0	Cosine of entry FP0.
	A7.L	Pointer to the stack with the return address removed.

Description: Returns cosine of value passed in FP0. Generates an invalid operation error if entry value is a NaN or an infinity.

10.2.3 Tangent

External label: .PTAN

Entry:	FP0	Argument in radians.
	A7.L	Pointer to return address on the stack.
Exit:	FP0	Tangent of entry FP0.
	A7.L	Pointer to the stack with the return address removed.

Description: Returns tangent of value passed in FP0. Generates invalid operation error if entry value is a NaN or an infinity.

10.2.4 Arctangent

External label: .PATN

Entry:	FP0	Argument.
	A7.L	Pointer to return address on the stack.
Exit:	FP0	Arctangent, in radians, of argument.
	A7.L	Pointer to the stack with the return address removed.

Description: Returns the arctangent of the value passed in FP0. Generates invalid operation error if entry value is a NaN. The result is in the range $-\pi/2$ to $+\pi/2$.

NOTE: ARCSIN(X) = ARCTAN(X/SQRT(1-SQR(X)))
ARCCOS(X) = π/2-ARCTAN(X/SQRT(1-SQR)X)))

10.2.5 Natural Logarithm

External label: .PLOG

Entry:	FP0	Argument.
	A7.L	Pointer to return address on the stack.
Exit:	FP0	Natural logarithm of entry FP0.
	A7.L	Pointer to the stack with the return address removed.

Description: Returns natural logarithm of value passed in FP0. Generates invalid operation error if entry value is a NaN, an infinity, or negative.

10.2.6 Exponential

External label: .PEXP

Entry:	FP0	Argument.
	A7.L	Pointer to return address on the stack.
Exit:	FP0	e raised to the power of the entry FP0.
	A7.L	Pointer to the stack with the return address removed.

Description: Returns result of e raised to the power of the value passed in FP0. Generates invalid operation error if argument is a NaN or an infinity. Generates overflow if argument is greater than approximately 11355.83. Generates underflow if argument is less than approximately -11400.19.

10.2.7 Power

External label: .PPWR

Entry:	FP0	The base.
	FP1	The exponent.
	A7.L	Pointer to return address on the stack.
Exit:	FP0	The result of raising the base to the exponent.
	FP1	Preserved.
	A7.L	Pointer to the stack with the return address removed.

Description: Returns result of raising entry value in FP0 by entry value in FP1. The result is determined by taking the exponential of the product of the power times the natural log of the base. Therefore, power can generate any of the errors that exponential, multiplication, and log can generate.

10.2.8 Round

External label: .PRND

Entry:	FP0	Argument.
	A7.L	Pointer to return address on the stack.
Exit:	FP0	Argument +.5 or -.5.
	D0.L	Result of rounding FP0.
	A7.L	Pointer to the stack with the return address removed.

Description: Round is performed by adding 0.5 to a positive argument or subtracting 0.5 from a negative argument, setting the rounding mode to round-to-zero, and then moving the argument to D0, all using floating point processor instructions. The original rounding mode is restored. An integer overflow error may be generated when the argument is moved to D0.

10.2.9 Truncate

External label: .PTRC

Entry:	FP0	Argument.
	A7.L	Pointer to return address on the stack.
Exit:	FP0	Preserved.
	D0.L	Result of truncating FP0.
	A7.L	Pointer to the stack with the return address removed.

Description: Truncate is performed by setting the rounding mode to round-to-zero and then moving the argument to D0, all using floating point processor instructions. The original rounding mode is restored. An integer overflow error may be generated when the argument is moved to D0.

10.2.10 Not-a-Number (NaN)

External label: .PNAN

Entry:	D0.L	The long integer to be placed in the significand of the NAN.
	A7.L	Pointer to return address on the stack.
Exit:	FP0	NaN result.
	D0.L	Preserved unless error; if error, contents destroyed.
	A7.L	Pointer to the stack with the return address removed.

Description: Returns a non-trapping NaN in FP0. Right justified in the significand of the NaN is the entry argument. A runtime error of 1070 is generated if the argument is zero.

10.3 REAL RUNTIME ROUTINES (FFP)

The fast floating point routines in the library PASCALIB.RO are written as subroutines which may be replaced by the user if desired. User-supplied routines must meet the same interface requirements as the furnished routines. All FFP routines must reside in the same contiguous block of memory because of word-sized branches between some routines.

This section lists the subroutines, their external labels, arguments, and entry and exit conditions. A separate paragraph describes exceptional conditions.

There are two general types of routines: functions and conversions. Most of the routines expect the primary argument to be in data register D7, and the secondary argument, if present, to be in register D6. For example, a floating point add operation must provide the source value in D6 and the destination value in D7. The square root function, however, is a 1-argument function; D7 will contain the argument, which will be replaced by the result. Most routines set the condition code as determined by the results of the operation involved. Most of the functions preserve the caller's register set. The amount of stack required for each routine is minimal.

The conversions -- along with the five primary functions add, subtract, multiply, divide, and square root -- return the highest accuracy possible. This is done by producing results as though to infinite precision during computation, and then rounding to single precision. When a result is exactly between two representable values in the format, rounding will occur up for positive values and down for negative values. This gives an effective 23.5 binary bits of precision, or approximately 7.2 decimal digits. However, there are cases where less precision is available, such as when cancellation occurs during the addition of oppositely-signed numbers which are very close in value. Also, anomalies such as "wobbling precision" inherent in conversions from floating-point to decimal may actually cause the return of eight full decimal digits of accuracy if the leading decimal digit is small. The transcendentals have varying precision, depending on arguments and the function invoked.

The FFP routines do not handle infinity or NaN's. Calculations or functions, which in the standard FP would have resulted in the creation of an infinity or a NaN, will cause a runtime error if FFP is enabled. In addition, the FFP routines do not maintain signed zeros (positive and negative); only positive zero is generated.

10.3.1 Sine

External label: .PQSIN

Entry:	D7.L	Argument in radians.
	A7.L	Pointer to return address on the stack.
Exit:	D7.F	Sine of argument.
	A7.L	Pointer to the stack with the return address removed.
	CCR	Z bit set if result is 0; N bit set if result is negative; V bit set if source value too large.

Description: Returns sine of source value. If source value is extremely large and little or no precision would result, zero is returned instead.

10.3.2 Cosine

External label: .PQCOS

Entry:	D7.L	Argument in radians.
	A7.L	Pointer to return address on the stack.
Exit:	D7.F	Cosine of argument.
	A7.L	Pointer to the stack with the return address removed.
	CCR	Z bit set if result is 0; N bit set if result is negative; V bit set if source value too large.

Description: Return cosine of source value. If source value too large, zero is returned.

10.3.3 Tangent

External label: .PQTAN

Entry:	D7.L	Argument in radians.
	A7.L	Pointer to return address on the stack.
Exit:	D7.F	Tangent of argument.
	A7.L	Pointer to stack with return address removed.
	CCR	Z bit set if result is 0; N bit set if result is negative; V bit set if source value too large.

Description: Return tangent of source value. If source value too large, zero is returned.

10.3.4 Arctangent

External label: .PQATAN

Entry:	D7.L	Argument.
	A7.L	Pointer to return address on the stack.
Exit:	D7.L	Arctangent, in radians, of argument.
	A7.L	Pointer to stack with return address removed.
	CCR	Z bit set if result 0; N bit cleared.

Description: Returns arctangent of source value in the range $-\pi/2$ to $\pi/2$.

NOTE: ARCSIN(X) = ARCTAN(X/SQRT(1-SQR(X)))
ARCCOS(X) = π/2-ARCTAN(X/SQRT(1-SQR)X)))

10.3.5 Natural Logarithm

External label: .PQLOG

Entry:	D7.L	Argument.
	A7.L	Pointer to return address on the stack.
Exit:	D7.L	Natural logarithm of entry argument.
	A7.L	Pointer to stack with return address removed.
	CCR	Z bit set if result 0; N bit set if result negative; V bit set if argument is negative.

Description: Returns natural logarithm of source value. A negative or 0 argument is illegal and causes a runtime error with the abort code set to invalid operation.

10.3.6 Exponential

External label: .PQEXP

Entry:	D7.L	Argument (base).
	A7.L	Pointer to return address on the stack.
Exit:	D7.L	e raised to the power of the base.
	A7.L	Pointer to stack with return address removed.
	CCR	Z bit set if result 0; N bit cleared.

Description: Returns result of raising e to the power of the base. Underflow will return 0. Overflow causes a runtime error.

10.3.7 Power

External label: .PQPWR

Entry:	D6.L	The exponent.
	D7.L	The base.
	A7.L	Pointer to return address on the stack.
Exit:	D6.L	Preserved.
	D7.L	Result of raising base to the exponent.
	A7.L	Pointer to stack with return address removed.
	CCR	Z bit set if result 0; N bit cleared.

Description: Returns the result of the base value taken to the power of the exponent. A negative or 0 base value is illegal and causes a runtime error. Underflow will return 0. Overflow causes a runtime error.

10.3.8 Round

External label: .PQRND

Entry:	D7.L	FP number.
	A7.L	Pointer to return address on the stack.
Exit:	D7.L	Rounded FP long word integer (2's complement).
	A7.L	Pointer to stack with return address removed.
	CCR	N bit set if result negative; Z bit set if result 0.

Description: Accepts FP value and replaces it by its rounded, signed 2's complement binary long word equivalent. Similar to .PQFPI, but rounds positive values by adding .5 and truncating; rounds negative values by subtracting .5 and truncating. Overflow causes a runtime error.

10.3.9 Test

External label: .PQTST

Entry:	D7.L	Destination number.
	A7.L	Pointer to return address on the stack.
Exit:	D7.L	Preserved.
	A7.L	Pointer to stack with return address removed.
	CCR	N bit set if result negative; Z bit set if 0.

Description: Sets the CCR as determined by the value in D7 for negative, positive, and zero.

10.3.10 Compare

External label: .PQCMP

Entry:	D6.L	Source number.
	D7.L	Destination number.
	A7.L	Pointer to return address on the stack.
Exit:	D6.L	Preserved.
	D7.L	Preserved.
	A7.L	Pointer to stack with return address removed.
	CCR	N and V bits set for proper arithmetic tests; Z bit set if result 0.

Description: Sets the CCR as determined by an arithmetic comparison of the source and destination values. The comparison is the result of the source taken from the destination. Neither argument is altered.

10.3.11 Absolute Value

External label: .PQABS

Entry:	D7.L	Argument number.
	A7.L	Pointer to return address on the stack.
Exit:	D7.L	Absolute value.
	A7.L	Pointer to stack with return address removed.
	CCR	N bit cleared; Z bit set if result 0.

Description: Return the absolute value of the argument.

10.3.12 Arithmetic Negate

External label: .PQNEG

Entry:	D7.L	Argument.
	A7.L	Pointer to return address on the stack.
Exit:	D7.L	Negated result,
	A7.L	Pointer to stack with return address removed.
	CCR	N bit set if result negative; Z bit set if result 0.

Description: Returns the negated value of the argument.

10.3.13 Addition

External label: .PQADD

Entry:	D6.L	Addend (source).
	D7.L	Adder (destination).
	A7.L	Pointer to return address on the stack.
Exit:	D6.L	Preserved.
	D7.L	Result of source added to destination.
	A7.L	Pointer to stack with return address removed.
	CCR	N bit set if result negative; Z bit set if result 0.

Description: Returns the sum of source and destination values. Underflow will return 0. Overflow causes a runtime error.

10.3.14 Subtraction

External label: .PQSUB

Entry:	D6.L	Subtrahend (source).
	D7.L	Minuend (destination).
	A7.L	Pointer to return address on the stack.
Exit:	D6.L	Preserved.
	D7.L	Result of source subtracted from destination.
	A7.L	Pointer to stack with return address removed.
	CCR	N bit set if result negative; Z bit set if result 0.

Description: Returns the result of subtracting the source from the destination; the sign of the source is inverted and the operation treated as an addition.

10.3.15 Multiplication

External label: .PQMUL

Entry:	D6.L	Multiplier (source).
	D7.L	Multiplicand (destination).
	A7.L	Pointer to return address on the stack.
Exit:	D6.L	Preserved.
	D7.L	Result of source multiplied by destination.
	A7.L	Pointer to stack with return address removed.
	CCR	N bit set if result negative; Z bit set if result 0.

Description: Returns result of multiplying the source by the destination. Underflow will return 0. Overflow causes a runtime error.

10.3.16 Division

External label: .PQDIV

Entry:	D6.L	Divisor (source).
	D7.L	Dividend (destination).
	A7.L	Pointer to return address on the stack.
Exit:	D6.L	Preserved.
	D7.L	Result of source divided into destination.
	A7.L	Pointer to stack with return address removed.
	CCR	N bit set if result negative; Z bit set if result 0.

Description: Returns result of dividing the source into the destination. Underflow will return 0. Overflow causes a runtime error. A zero divisor is illegal and causes a runtime error.

10.3.17 Square Root

External label: .PQSQRT

Entry:	D7.L	Argument.
	A7.L	Pointer to return address on the stack.
Exit:	D7.L	Square root of argument.
	A7.L	Pointer to stack with return address removed.
	CCR	Bit N is cleared and bit Z is set if result 0. A negative value is illegal and causes a runtime error.

Description: Returns the square root of the argument.

10.3.18 Division with Remainder

External label: .PQREM

Entry:	D6.L	Divisor.
	D7.L	Dividend.
	A7.L	Pointer to return address on the stack.
Exit:	D6.L	Preserved.
	D7.L	Remainder.
	A7.L	Pointer to stack with return address removed.
	CCR	N bit set if result negative; Z bit set if result 0.

Description: Similar to .PQDIV but there is no overflow condition.

10.3.19 Conversion of Floating Point to Integer (Truncate)

External label: .PQFPI

Entry:	D7.L	FP number.
	A7.L	Pointer to return address on the stack.
Exit:	D7.L	Fixed point long word integer (2's complement).
	A7.L	Pointer to stack with return address removed.
	CCR	N bit set if result negative; Z bit set if result 0.

Description: Accepts FP value and replaces it by its signed 2's complement binary long word equivalent. Range provided is -2,147,483,649 < value < +2,147,483,648. If magnitude of input is larger than that allowed by 32-bit signed binary value, overflow will occur. Results of over 24-bit integer magnitude are imprecise and have low end zeros.

10.3.20 Conversion of Integer to Floating Point

External label: .PQIFP

Entry:	D7.L	Fixed point long word integer (2's complement).
	A7.L	Pointer to return address on the stack.
Exit:	D7.L	FP number.
	A7.L	Pointer to stack with return address removed.

Description: Accepts a signed 2's complement binary long word value and replaces it with a single-precision FP value. Integers of more than 24 bits in size will be rounded and imprecise.

10.3.21 Read Real

External label: .PQRDR

Entry:	A0.L	Address of Pascal file pointer.
	A1.L	Address of real number.
	A7.L	Pointer to return address on the stack.
Exit:	A0.L	Preserved.
	A1.L	Preserved.
	A7.L	Pointer to stack with return address removed.

Description: Reads a real number from a text file and stores it at the specified location.

10.3.22 Write Real

External label: .PQWRR

Entry:	D0.W	Width of field for real number.
	D1.W	Width of field for fractional part; if D1 negative, number written in exponential format.
	D7.L	Real number.
	A0.L	Address of Pascal file pointer.
	A7.L	Pointer to return address on the stack.
Exit:	D0.W	Contents destroyed.
	D1.W	Contents destroyed.
	D7.L	Contents destroyed.
	A0.L	Preserved.
	A7.L	Pointer to stack with return address removed.

Description: Writes a real number to a text file in fixed point or floating point format.

10.3.23 Exceptional Conditions

The following list details all unusual conditions possible during fast floating point operations and identifies the results for each case.

Underflow

Underflow occurs when a result is smaller than the smallest representable value supported. When this occurs a positive zero result is returned with no abnormal condition signaled.

Overflow

When an operation exceeds the largest positive or negative normalized value handled, an overflow condition occurs. Overflow causes a runtime error with abort code 2002.

Divide by Zero

Division by zero is invalid and will cause a runtime error with abort code 2008. (Division of zero by zero yields abort code 2400.)

Negative Square Root

A negative square root other than zero is invalid and will cause a runtime error with abort code 2001.

Invalid ASCII Conversion to Float

When converting a character string to float and an invalid pattern is detected, a runtime error will occur with abort code 1032.

Sine/Cosine/Tangent Argument Too Large

If the input argument to any sine, cosine, or tangent function will result in less than 5 bits of precision because of excessive magnitude, zero is returned with the V bit set in the condition code register.

Negative X for X**Y Power Function

If a negative value is used to take to a floating power, a runtime error will occur with abort code 2001.

Negative X for Log(X)

If a negative logarithm is attempted, a runtime error will occur with abort code 2001.

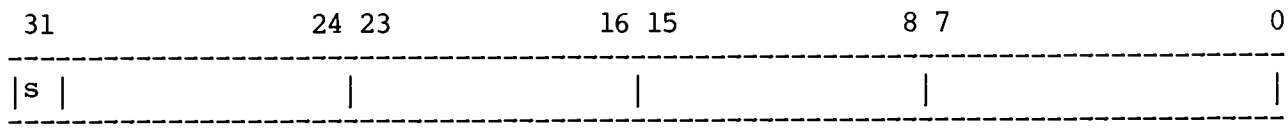
CHAPTER 11

INTERNAL REPRESENTATION OF DATA

11.1 INTERNAL REPRESENTATION

The floating point data in this section is generally valid for both standard floating point and fast floating point. Exceptions are noted.

Integer:

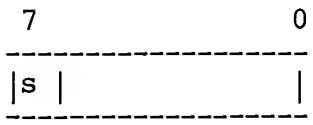


Size: 4 bytes

Format: Signed two's-complement

Range: -2,147,483,648 to 2,147,483,647

for an integer subrange type within the range -128 to 127, inclusive:

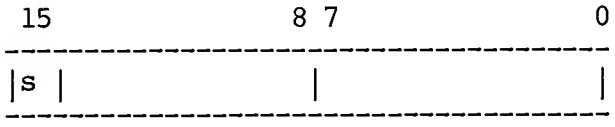


Size: 1 byte

Format: Signed two's-complement

Range: -128 to 127

for an integer subrange type that extends outside the range -128 to 127, inclusive, but is within the range -32,768 to 32,767, inclusive:

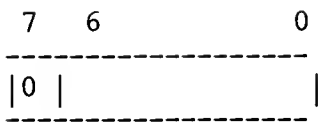


Size: 2 bytes

Format: Signed two's-complement

Range: -32,768 to 32,767

Character:

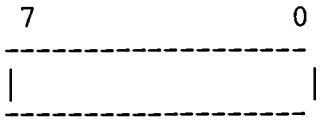


Size: 1 byte

Format: 7-bit ASCII

Range: 0 to 127

Boolean:

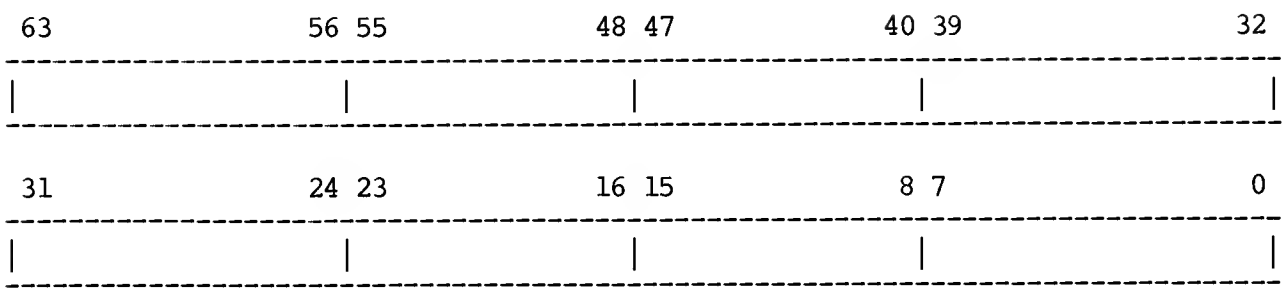


Size: 1 byte

Values: 0 = False

1 = True

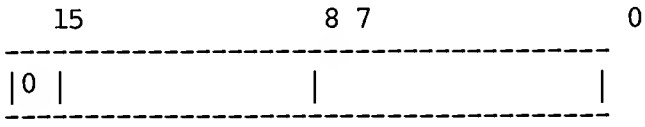
Set:



Size: 8 bytes

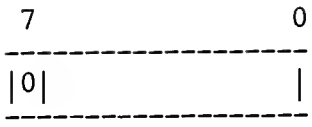
Range: Up to 64 elements

Enumerated Scalar:



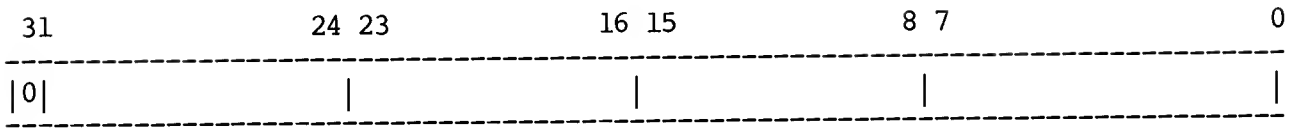
Size: 2 bytes

Representation: 0 to 32,767



Size: 1 byte

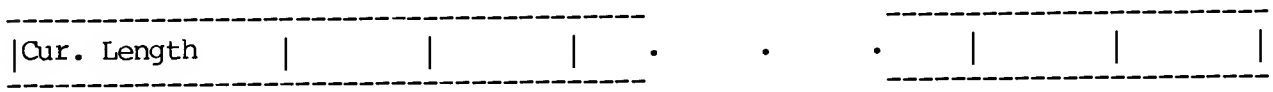
Representation: 0 to 127



Size: 4 bytes

Representation: 0 to 2,147,483,647

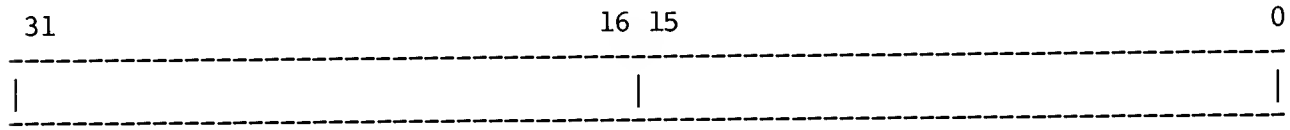
String:



Size: 2 to 32766 bytes

Representation: Current-length word and
0 to 32764 ASCII characters

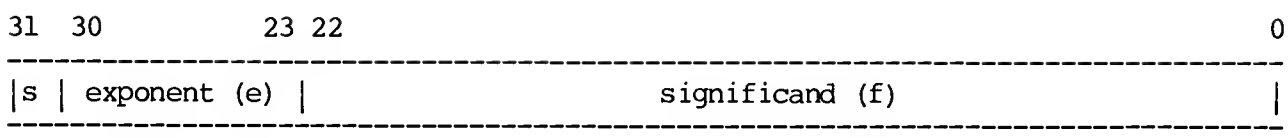
Pointer:



Size: 4 bytes

Range: 0 to $2^{32} - 1$

Real:



Length in bits	32
Interpretation of sign:	
positive	0
negative	1
Normalized Numbers:	
interpretation of e	unsigned integer
bias of e	127
range of e	0 < e < 255
interpretation of significand	1.f
relation to representation of real numbers	(-1)^s x 2^(e-127) x 1.f

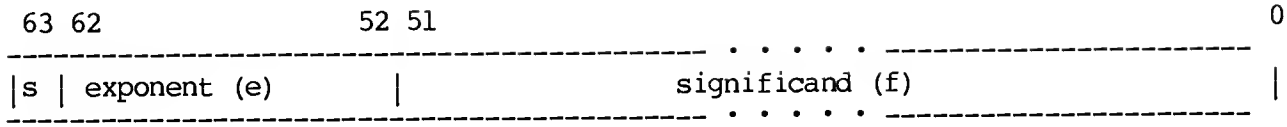
NOTE

Significand lies in the range 1.0 ≤ significand < 2.0, with the integer part implicit.

Signed Zeroes:	
e =	0
f =	0
Reserved Operands:	
Denormalized numbers:	
e =	0
bias of e	126
interpretation of significand	0.f
range of f	nonzero
relation to representation of real numbers	(-1)^s x 2^-126 x 0.f
Signed Infinities:	
e =	255
f =	0
Not-a-Numbers:	
e =	255
f =	nonzero
interpretation of significand	don't care

Ranges:	Standard FP	FFP
maximum positive normalized	3.4 x 10^38	9.2 x 10^18
minimum positive normalized	1.2 x 10^-38	5.4 x 10^-20
minimum positive denormalized	1.4 x 10^-45	--
maximum negative normalized	-1.2 x -10^38	-2.7 x 10^-20
minimum negative normalized	-3.4 x -10^-38	-9.2 x 10^18

Dreal (standard FP only):



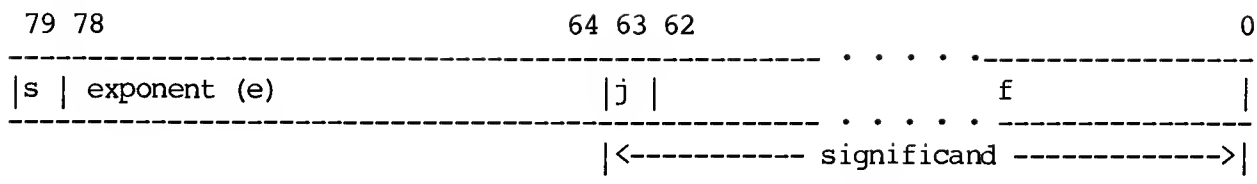
Length in bits	64
Interpretation of sign:	
positive	0
negative	1
Normalized Numbers:	
interpretation of e	unsigned integer
bias of e	1023
range of e	0 < e < 2047
interpretation of significand	1.f
relation to representation of real numbers	(-1)^s x 2^(e-1023) x 1.f

NOTE

Significand lies in the range 1.0 ≤ significand < 2.0, with the integer part implicit.

Signed Zeroes:	
e =	0
f =	0
Reserved Operands:	
Denormalized numbers:	
e =	0
bias of e	1022
interpretation of significand	0.f
range of f	nonzero
relation to representation of real numbers	(-1)^s x 2^-1022 x 0.f
Signed Infinities:	
e =	2047
f =	0
Not-a-numbers:	
e =	2047
f =	nonzero
interpretation of significand	don't care
Ranges:	
maximum positive normalized	1.8 x 10^307
minimum positive normalized	2.2 x 10^-308
minimum positive denormalized	4.9 x 10^-324

Xreal (standard FP only):



Length in bits	80
Interpretation of sign:	
positive	0
negative	1
Normalized Numbers:	
interpretation of e	2's complement integer
bias of e	0
range of e	-16384 <= e < 16383
interpretation of significand	j.f
relation to representation of real numbers	(-1)^s x 2^e x j.f

NOTE

Significand lies in the range 1.0 $\leq$ significand < 2.0, with the integer part explicit.

Signed Zeroes:	
e =	-16384 ($4000)
significand =	0
Reserved Operands:	
Denormalized numbers:	
e =	-16384
bias of e	0
interpretation of significand	0.f
range of f	nonzero
relation to representation of real numbers	(-1)^s x 2^-16384 x 0.f
Signed Infinities:	
e =	16383 ($3FFF)
significand =	0
Not-a-numbers:	
e =	16383 ($3FFF)
significand =	nonzero
interpretation of significand	don't care
Ranges:	
maximum positive normalized	6 x 10^4931
minimum positive normalized	8 x 10^-4933
minimum positive denormalized	9 x 10^-4952

11.2 DEFINITIONS

Biased Exponent - The sum of the exponent and a constant (Bias) chosen to make the biased exponent's range non-negative.

Binary Floating Point Number - A bit string characterized by three components: a sign, a signed exponent, and a significand. Its numerical value, if any, is the signed product of its significand and two raised to the power of its exponent. A bit string is not always distinguished from a number it may represent.

Denormalized - The exponent is the format's minimum, the explicit or implicit leading bit is a zero, and the number is not normal zero. To denormalize a binary floating point number means to shift its significand right while incrementing its exponent, until it is a denormalized number.

Exponent - That component of a binary floating point number which signifies the power to which two is raised in determining the value of the represented number. Occasionally, the exponent is called signed or unbiased exponent.

FFP - An abbreviation for 'fast floating point'.

FP - An abbreviation for 'floating point'.

Fraction - The field of the significand that lies to the right of its implied binary point.

Infinities - Infinities are represented by having all exponent bits on and significand bits off. The sign bit determines the difference between a plus or minus infinity.

NaN - Not a Number; a bit representation which indicates that the number is not a valid floating point number. NaN's may be user-generated, for example, when initializing areas of memory to indicate that no valid floating point number has been stored there.

Normalized - If the number is nonzero, shift its significand left while decrementing its exponent until the leading significand bit becomes one; the exponent is regarded as if its range were unlimited. If the significand is zero, the number becomes normal zero. Normalizing a number does not change its sign.

Normal Zero - The exponent is the format's minimum and the significand is zero. Normal zero may have either a positive or negative sign. Only the extended format has any unnormalized zeroes.

Significand - That component of a binary floating point number which consists of an explicit or implicit leading bit to the left of its binary point and a fraction field to the right of the binary point.

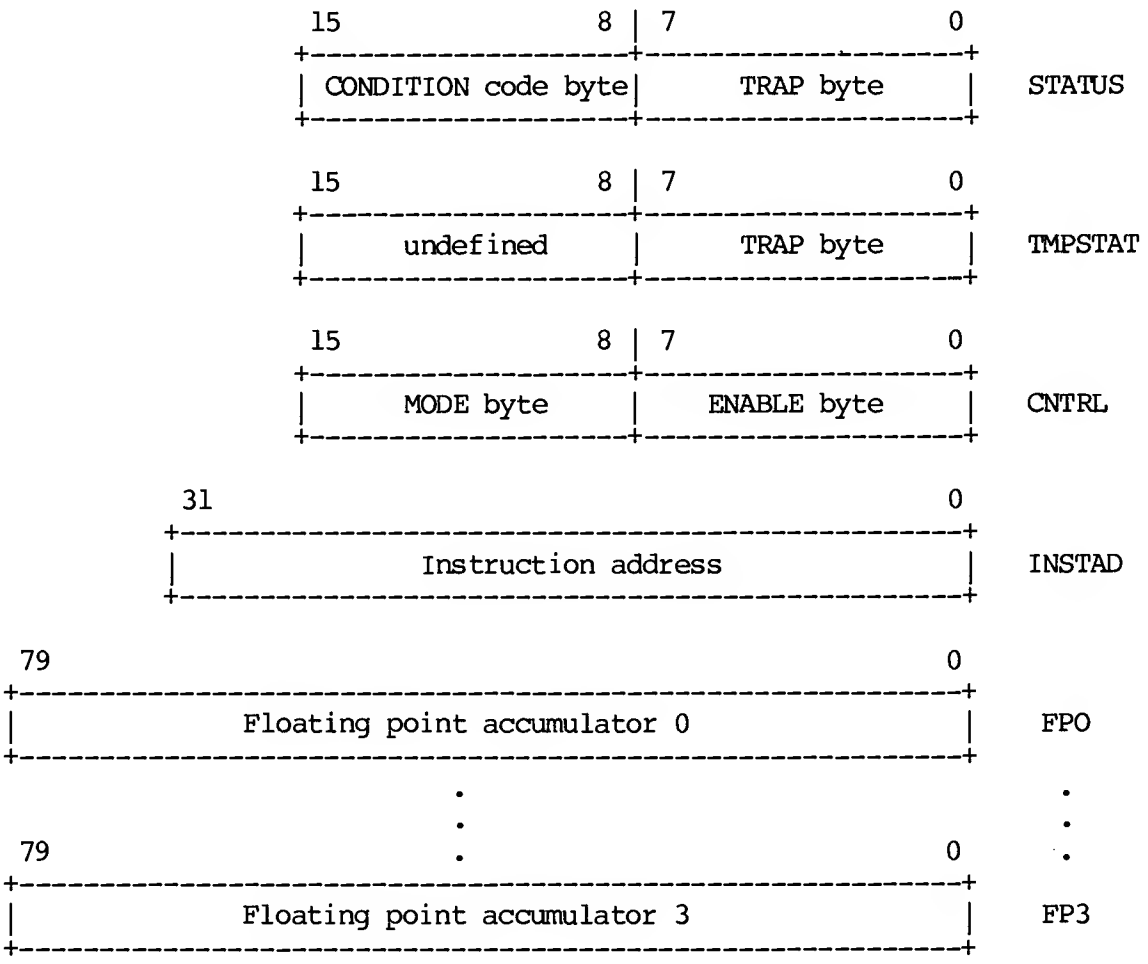
Unnormalized - The exponent is greater than the extended format's minimum and the explicit leading bit is zero. If the significand is zero, this is an unnormalized zero.

APPENDIX A

STANDARD FLOATING POINT PROCESSOR

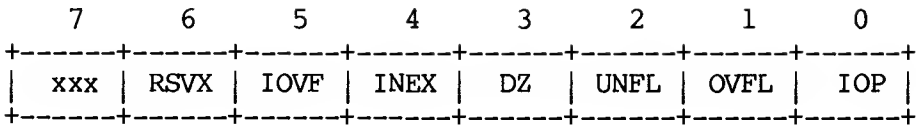
A.1 PROGRAMMER'S MODEL

The data in this appendix applies to the Motorola standard floating point, M68341.



The standard floating point processor has eight registers available to the programmer. The four floating point accumulators (FP0, FP1, FP2, and FP3) contain floating point numbers in internal format. The instruction address register contains the address of the floating point instruction currently being executed. The control register is initialized by the programmer to specify the modes of calculation and of trapping on errors. The status register is set by the floating point processor and may be examined by the programmer. The temporary status is used to hold the status of the last operation only. The bytes making up the status and control registers are further defined in the following paragraphs.

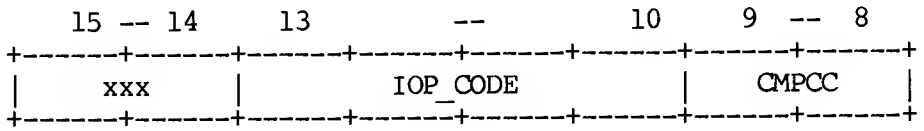
A.1.1 TRAP Status Byte



The bits in the trap status byte are set if any errors have occurred. Note that each bit of the trap status byte must be reset by the caller. The FP processor only writes 1 bits into the status byte, and never clears existing bits. This is done so that a long calculation can be completed with the error status checked once at the end. Note that the bits in the TRAP status byte are in the same bit positions as the corresponding bits in the ENABLE mode byte.

Bit 0	Invalid operation
Bit 1	Overflow
Bit 2	Underflow
Bit 3	Divide-by-zero
Bit 4	Inexact result
Bit 5	Integer overflow
Bit 6	Reserved exponent value (minimum or maximum exponent) seen as an input operand
Bit 7	Unused, reserved

A.1.2 CONDITION Code Status Byte



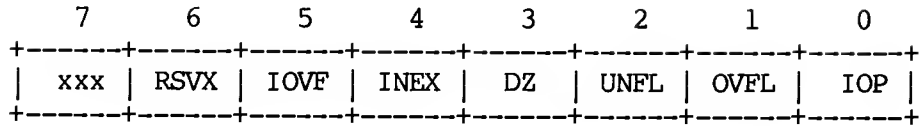
This result status byte contains the result of a floating point compare instruction and also the type of an invalid operation error, if one occurred.

Bits 8-9 Condition codes from FCMP instruction (CMPCC)
- 00 = Equal
- 01 = Less than
- 10 = Greater than
- 11 = Unordered

Bits 10-13 Invalid operation code (IOP_CODE)
This field is set when an invalid operation occurs and the IOP bit is set in the TRAP status byte.
- 0 = No IOP error.
- 1 = Square root of a negative number, infinity in projective mode, or a not normalized number.
- 2 = (+infinity) + (-infinity) in affine mode.
- 3 = Tried to convert NaN to binary integer.
- 4 = In division: 0/0, infinity/infinity or divisor is not normalized and the dividend is not zero and is finite.
- 5 = One of the input arguments was a trapping NaN.
- 6 = Unordered condition tested by predicate other than equal or not-equal.
- 7 = Projective closure use of +/- infinity.
- 8 = 0 * infinity.
- 9 = In REM <ea> is zero or not normalized or FPn is infinite.
- 10 = Value of 'k' for BINDEC or 'p' for DECBIN is out of range.
- 11 = Tried to MOV a single denormalized number to a double destination.
- 12 = Tried to return an unnormalized number to single or double (invalid result).
- 13 = Illegal instruction

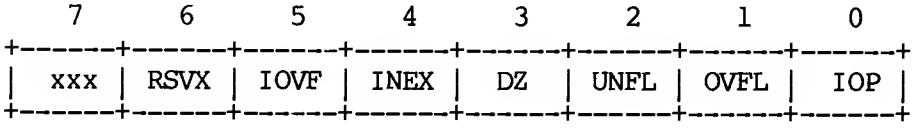
Bits 14-15 Unused, reserved

A.1.3 Temporary Trap Status Byte (TMPSTAT)



The bits in the temporary status byte are identical to those in the TRAP status byte except they only represent the status of the last operation. They are cleared at the start of each operation. They can be used by the trap handler to determine the cause of a trap. At the end of the operation, they are ORed into the TRAP status byte to create the sticky bits that can be used to determine the status of a string of operations.

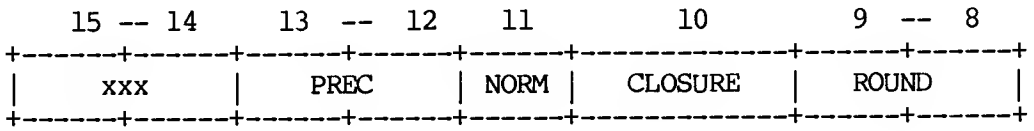
A.1.4 ENABLE Byte



The programmer may set a one in any bit to enable a trap on the corresponding error condition.

Bit 0	Invalid operation
Bit 1	Overflow
Bit 2	Underflow
Bit 3	Divide-by-zero
Bit 4	Inexact result
Bit 5	Integer overflow
Bit 6	Reserved exponent value (maximum or minimum exponent) seen as an input operand
Bit 7	Unused, reserved

A.1.5 MODE Byte



The programmer sets bits in this byte to control the calculation modes as defined below.

Bits 8-9 — Rounding mode (ROUND)
- 00 = Round to nearest
- 01 = Round to zero
- 10 = Round to plus infinity
- 11 = Round to minus infinity

Bit 10 — Affine/projective mode (CLOSURE)
- 0 = Projective closure
- 1 = Affine closure

Bit 11 — Normalizing mode select (NORM)
- 1 = Normalize denormalized numbers while converting to internal format. (Normalizing mode)
- 0 = Do not normalize denormalized operands before an operation. (Warning mode)

Note: Unnormalized numbers are not affected by bit 11.

Bits 12-13 — Rounding precision select (PREC)
- 00 = Round to extended
- 01 = Round to single
- 10 = Round to double
- 11 = Unused, reserved

Bits 14-15 — Unused, reserved

A.2 FLOATING POINT OPERATIONS

A.2.1 Memory Data Format Descriptors

Dyadic operations may be applied to operands of differing formats, since all operands are converted to internal (extended) format for the actual calculation. The floating point format of an operand stored in memory is identified as follows:

S = Single
D = Double
X = Extended

Three sizes of binary integers may also be specified by:

B = Byte (8 bit)
W = Word (16 bit)
L = Long (32 bit)

Finally, the conversion from binary floating point to decimal (BCD) representation will be provided and is indicated by an FMOVE instruction with the data type:

P = Decimal format

A.2.2 Assembler Instruction Format

The assembler instruction format is designed to follow the current M68000 family assembler conventions as closely as possible.

The format is:

```
        <label>        <mne>.<siz>    <addr>
```

where:

```
    <label>    ::=  Symbolic label
    <mne>      ::=  Instruction mnemonic
    <siz>      ::=  S | D | X | B | W | L | P
    <addr>     ::=  <reg> | <ea> | <src>,<dst>
  <src>,<dst>  ::=  <reg>,<reg> | <ea>,<reg> | <reg>,<ea>
    <ea>       ::=  MC68000/MC68010 addressing mode
    <reg>      ::=  <facc>
                    |  CNTRL              Control register
                    |  STATUS             Status register
                    |  INSTAD             Instruction address
                    |  TMPSTAT            Current status
    <facc>     ::=  FP0 | FP1 | FP2 | FP3  FP accumulators
```

A.2.3 Move Instructions

The mnemonic 'FMOVE' is used to transfer data into and out of the floating point processor. It has the auxiliary effect of converting between the various supported formats. Some examples of the FMOVE instruction are:

```
FMOVE.<siz>     <ea>,FPn
FMOVE.<siz>     FPn,<ea>
FMOVE           FPn,FPm
FMOVE           <ea>,CNTRL
FMOVE           CNTRL,<ea>
FMOVE           <ea>,STATUS
FMOVE           STATUS,<ea>
FMOVE           INSTAD,<ea>
FMOVE           <ea>,INSTAD
```

It should be noted that, although the data type identifier is syntactically associated with the instruction, in the case of floating point instructions, the data type is actually associated with the <ea> field. All numbers in the floating point accumulators are represented in extended format; the result of any arithmetic operation is also in extended format. The data type descriptor really describes the memory format of the data. Any move between a floating point accumulator and memory implies a conversion between extended format and the specified memory format, with a possible rounding operation.

A.2.4 Arithmetic Operations

The dyadic arithmetic operations expect one operand to be in a floating point accumulator. The other operand may be in a floating point accumulator, in memory, or in an MC68000/MC68010 data register. The result is left in a floating point accumulator. Monadic operations work upon an operand in a floating point accumulator and leave the result in the same accumulator. The supported operations are:

Dyadic

FADD.B	W	L	S	D	X	<src>,FPn	Addition
FSUB.B	W	L	S	D	X	<src>,FPn	Subtraction
FMUL.B	W	L	S	D	X	<src>,FPn	Multiplication
FDIV.B	W	L	S	D	X	<src>,FPn	Division
FREM.B	W	L	S	D	X	<src>,FPn	Remainder
FCMP.B	W	L	S	D	X	<src>,FPn	Compare

Monadic

FINT	FPn	Integer-part
FABS	FPn	Absolute value
FSQRT	FPn	Square root
FNEG	FPn	Negate
FNOP		Wait for completion of previous operation

A.2.5 Floating Point Compare Instructions

Comparisons may be performed between two floating point numbers and the result returned as condition code bits in the status register. The floating point compare is a dyadic operation which compares the contents of a floating point accumulator with another floating point accumulator or with the contents of a MC68000/MC68010 effective address. The compare instruction has the form:

```
FCMP.B|W|L|S|D|X        <ea>,FPn        Compare FP accumulator
```

which sets the condition codes according to FPn - <ea>, and

```
FCMP            FPn,FPm         Compare FP accumulators
```

which sets the condition codes according to FPm - FPn.

A.2.6 Test for Special Value Instructions

The contents of a floating point accumulator or memory operand may be tested for the five kinds of floating point values. The operation performed is a test for equality between the contents of the effective address and a particular special value. These instructions are:

```
ISNAN.S|D|X             <src>       Is it a NaN?
ISZERO.S|D|X            <src>       Is it normal zero?
ISINF.S|D|X             <src>       Is it infinity?
ISNORM.S|D|X            <src>       Is it a normalized,
                                        non-zero number?
ISNNORM.S|D|X           <src>       Is it not normalized?
```

The condition code is set so that 'equal' means the test is true, and 'not equal' means the test is false.

A.2.7 Conditional Branch Instructions

The result of a compare or special value test may be accessed in two ways. Branching on the compare condition codes is possible, using:

FBEQ	<ea>	Equal
FBNE	<ea>	Not equal
FBGE	<ea>	Greater or equal
FBGT	<ea>	Greater
FBLE	<ea>	Less or equal
FBLT	<ea>	Less
FBUND	<ea>	Unordered
FBORD	<ea>	Ordered

where the <ea> is limited to relative addressing.

Alternatively, the instructions

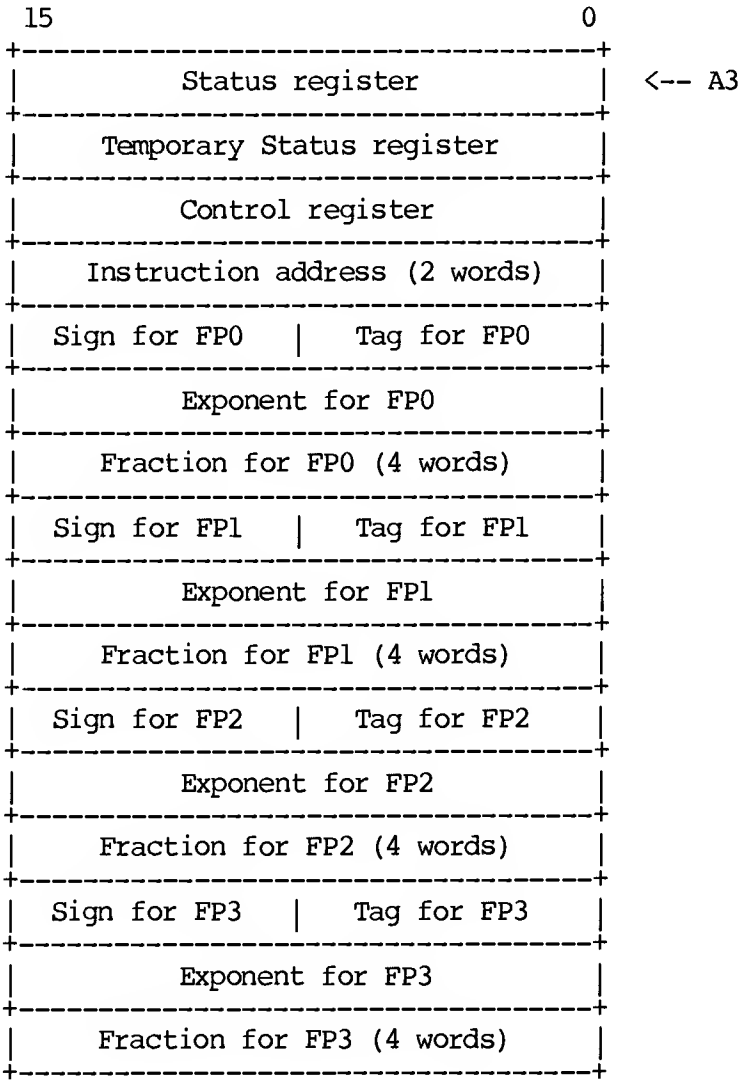
FSEQ	<dest>	Set if equal
FSNE	<dest>	Set if not equal
FSLE	<dest>	Set if less or equal
FSLT	<dest>	Set if less than
FSGE	<dest>	Set if greater or equal
FSGT	<dest>	Set if greater than
FSUND	<dest>	Set if unordered
FSORD	<dest>	Set if ordered

will store a word of all one's (for true) or all zero's (for false) at the effective address, depending on whether the specified condition is true or false. Standard MC68000/MC68010 instructions may then be used to test the result of the comparison. The <ea>'s allowed are the same as the ones for the MC68000/MC68010 'Scc' instructions.

A.3 SOFTWARE IMPLEMENTATION

A.3.1 Floating Point Register Block

The FP software maintains a block of 29 words in RAM, which simulates the user-visible registers of the floating point processor. The 'front end' F-line trap routine will set up register A3 to point to the register block. The organization of the FP register block in terms of 16-bit words is shown below.

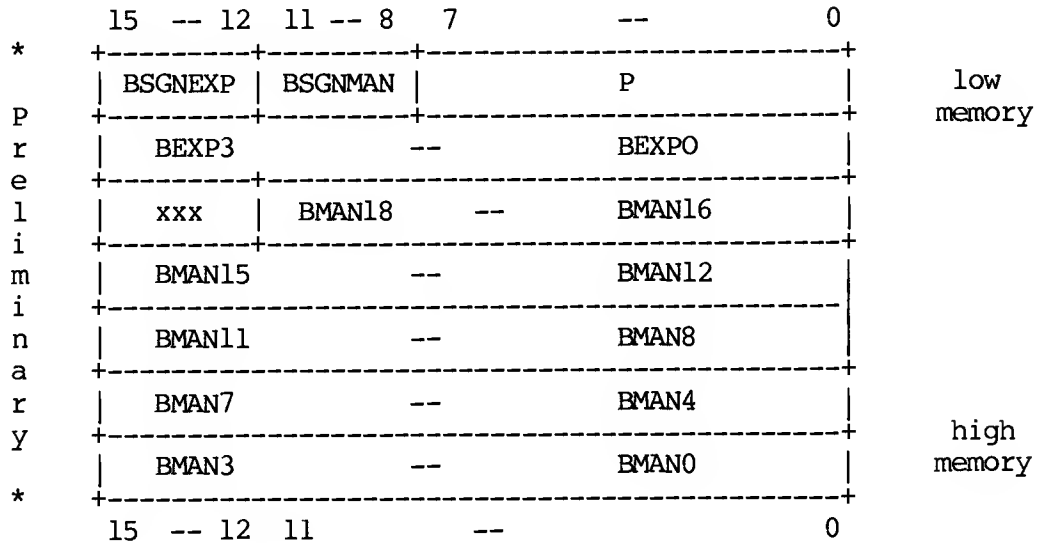


A.3.2 Binary-Decimal Conversions

The BCD representation of the contents of a floating point accumulator may be stored to memory using:

```
FMOVE.P        FPn,<ea>
FMOVE.P        <ea>,FPn
```

which converts the contents of floating point accumulator 'n' to seven words in BCD format as follows:



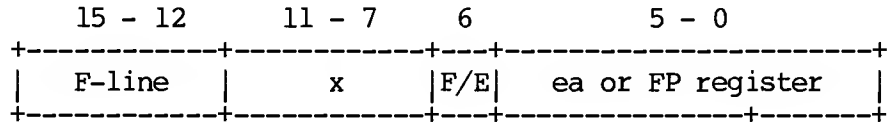
where:

BSGNEXP	Sign of exponent $0 = Positive $A = Number is +infinity (other digits zero) $B = Number is -infinity (other digits zero) $C = Number is a NaN (other digits zero) $F = Negative
BEXPn	Nth BCD digit of exponent
BSGNMAN	Sign of mantissa $0 = Positive $F = Negative
BMANn	Nth BCD digit of mantissa
P	Binary integer giving the number of BCD digits of mantissa to the right of the decimal place. The range of P is: 0 <= P <= 19 on input.
xxx	Unused, reserved

Note that non-numeric values are represented by the special codes $A, $B, and $C in the sign of the exponent; 'P' will be equal to zero for all non-numeric values. Infinities will have all other BCD digits set to zero. A NaN will have the binary error address stored right-justified in the mantissa field.

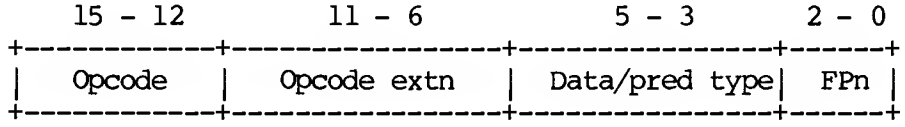
A.3.3 Software Floating Point Instruction Format

The first word of a floating point instruction will be an F-line opcode which will trap to the F-line trap handler. This word will contain fields that define the effective address for the MC68000/MC68010. The fields are:



F-line	F-line emulator opcode
F/E	FPn/effective address bit 0 = Next field contains FP register number 1 = Next field contains MC68000/MC68010 effective address
ea	68000/68010 effective address field, or FP register
x	Unused, reserved

It is followed by a word which contains the floating point instruction having the format:



where the fields are defined as:

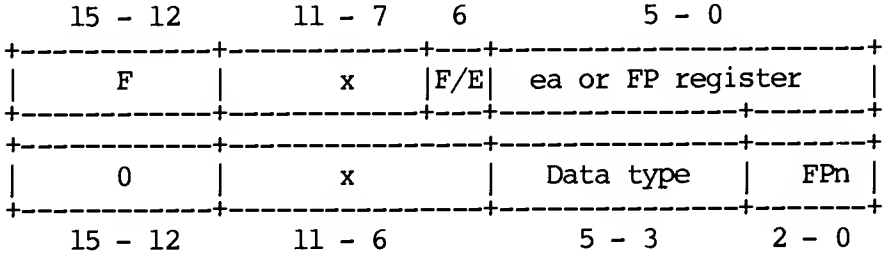
Opcode	FP operation code
0 =	FMOVE into FP processor
1 =	FMOVE out of FP processor
2 =	FMOVE special registers in
3 =	FMOVE special registers out
4 =	Reserved
5 =	Reserved
6 =	Arithmetic dyadic operations
7 =	Arithmetic monadic operations
8 =	Special value tests (ISNAN, etc.)
9 =	Conditional branches
A =	Set byte on condition
B-F =	Unused, reserved

Opcode extn	Extension field - use depends on opcode.
Data/pred type	Type of operand in memory or predicate for branch instruction.
FPn	Destination or source FP register.

A.3.3.1 Operation Code Field Details

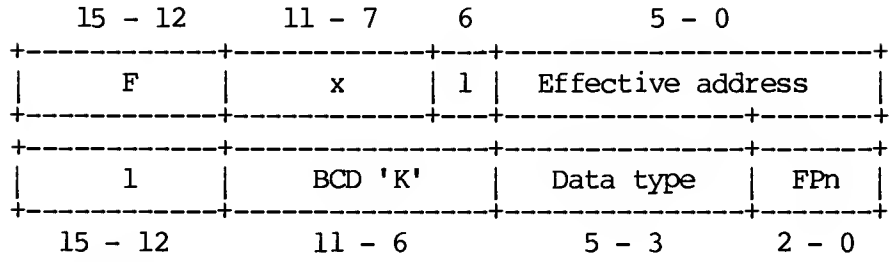
Each operation code defines a class of floating point instructions. The following paragraphs specify the detailed definition of the fields required in each class of instructions. The two 16-bit words which make up each instruction are diagrammed.

A.3.3.1.1 FMOVE Memory Operand into a Floating Point Register



F/E	FPn/effective address bit 0 = ea field contains FP register number 1 = ea field contains MC68000/MC68010 effective address
Effective address	Source address or FP register
Data type	Type of operand in memory 0 = S (single FP) 1 = D (double FP) 2 = X (extended FP) 3 = B (byte integer) 4 = W (word integer) 5 = L (long integer) 6 = P (BCD)
FPn	Destination floating point register
x	Unused, reserved

A.3.3.1.2 FMOVE Floating Point Register into a Memory Operand



Effective address — Destination address

FPn — Source floating point register

Data type — Type of operand in memory

0 = S (single FP)
1 = D (double FP)
2 = X (extended FP)
3 = B (byte integer)
4 = W (word integer)
5 = L (long integer)
6 = P (BCD) w/ K static
7 = P (BCD) w/ K dynamic

BCD Signif (K) — A two's complement integer value in the ranges:

(-16) <= K <= (-1) or (+1) <= K <= (+17)

When K is negative in the range (-16) <= K <= (-1), it represents the desired number of decimal digits to the right of the decimal point; use K negative when the BCD string is destined to be converted to a fixed point format (Pascal's F:x:y format).

When K is positive in the range (1) <= K <= (17), it represents the desired number of significant decimal digits; use K positive when the BCD string is destined to be converted to a floating point format (Pascal's F:x format).

When K is outside of the allowed ranges specified above [i.e., K = 0, K < (-16), or K > (17)], the result BCD string will be calculated using the nearest valid value of K [when K = 0, the resulting BCD string will be calculated with K = 1]. Note that an IOP = 10 will accompany the BCD string result in these cases.

When data type specifies P (BCD) w/ K static (110):

nnnnnn = Two's complement immediate value of K in the ranges specified above.

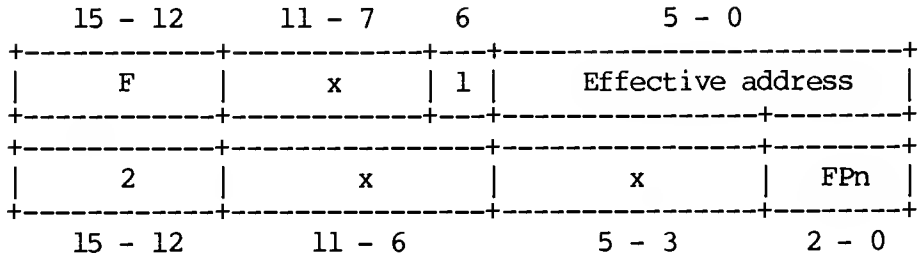
When data type specifies P (BCD) w/ K dynamic (111):

nnnRRR = Number of data register which contains the two's complement immediate value of K in the ranges specified above.

FPn — Floating point accumulator

x — Unused, reserved

A.3.3.1.3 FMOVE Memory Operand into a Special Floating Point Register

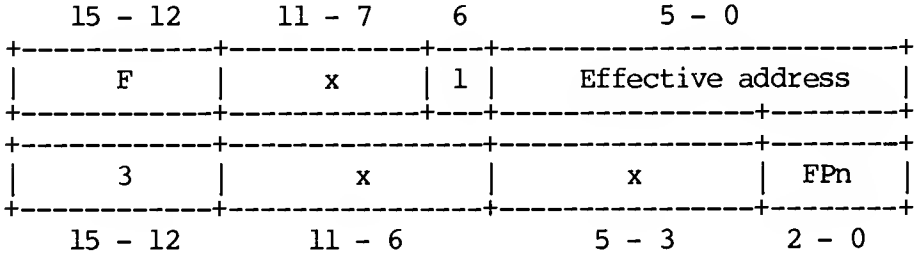


Effective address — Source address

FPn — Destination special floating point register
- 0 = CNTRL
- 1 = STATUS
- 2 = TMPSTAT
- 3 = INSTAD (Instruction address)
- 4-7 = Unused, reserved

x — Unused, reserved

A.3.3.1.4 FMOVE Special Floating Point Register out to a Memory Operand

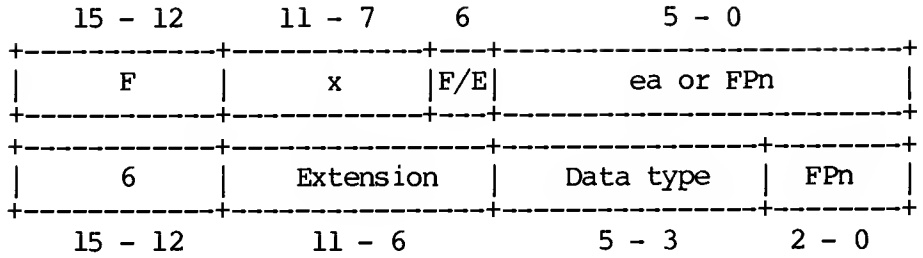


Effective address — Destination address

FPn — Source special floating-point register
- 0 = CNTRL
- 1 = STATUS
- 2 = TMPSTAT
- 3 = INSTAD (Instruction address)
- 4-7 = Unused, reserved

x — Unused, reserved

A.3.3.1.5 Arithmetic Dyadic Operations



ea or FPn — Source effective address or floating point register

F/E — Source is ea or FPn
0 = FP register
1 = MC68000/MC68010 effective address

Extension — Arithmetic operation
0 = Add
1 = Subtract
2 = Multiply
3 = Divide
4 = Remainder
5 = Compare
6-3F = Unused, reserved

Data type — Type of operand in memory
0 = S (single FP)
1 = D (double FP)
2 = X (extended FP)
3 = B (byte integer)
4 = W (word integer)
5 = L (long integer)
6 = Illegal *
7 = Illegal *

* - These two data types would signify an instruction of the form:

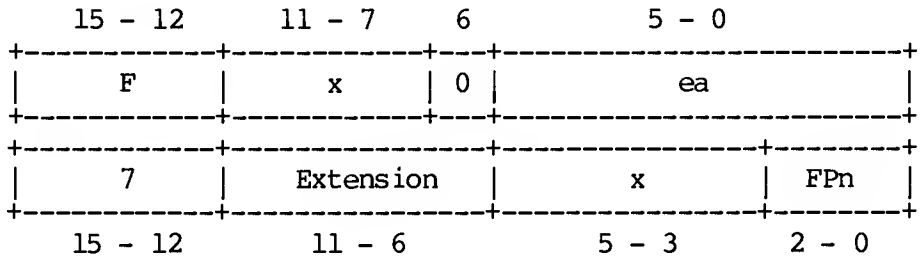
F(arith).P <ea>,FPn

which is not allowed; an IOP = 13 would be produced.

FPn — Destination floating point register

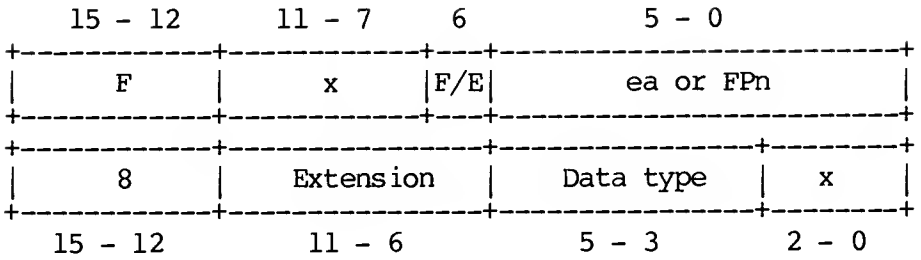
x — Unused, reserved

A.3.3.1.6 Arithmetic Monadic Operations



ea field	Source FP register (same as destination)
Extension	Arithmetic operation 0 = Integer-part-of 1 = Absolute value 2 = Square root 3 = Negate 4 = No-operation 5 - 3F = Unused, reserved
FPn	Source and destination floating point register
x	Unused, reserved

A.3.3.1.7 Special Value Tests



ea or FPn — Source effective address or floating point register

F/E — Source is ea or FPn
- 0 = FP register
- 1 = MC68000/MC68010 effective address

Extension — Special value to test for:
- 0 = Normal zero
- 1 = Not-a-number
- 2 = Infinity
- 3 = Normalized
- 4 = Not normalized
- 5 - 3F = Unused, reserved

Data type — Type of operand in memory
- 0 = S (single FP)
- 1 = D (double FP)
- 2 = X (extended FP)
- 3 = B (byte integer)
- 4 = W (word integer)
- 5 = L (long integer)
- 6 = Illegal *
- 7 = Illegal *

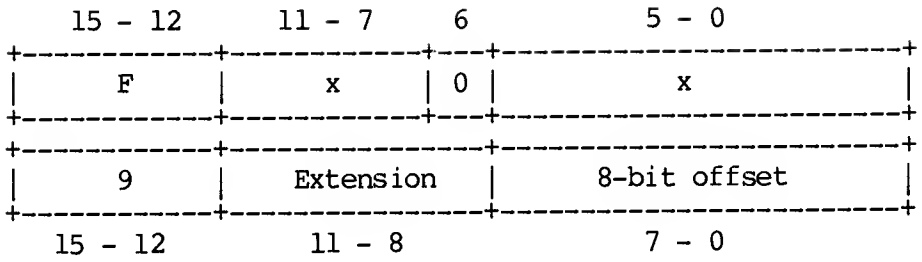
* - These two data types would signify an instruction of the form:

IS(type).P <ea>

which is not allowed; an IOP = 13 would be produced.

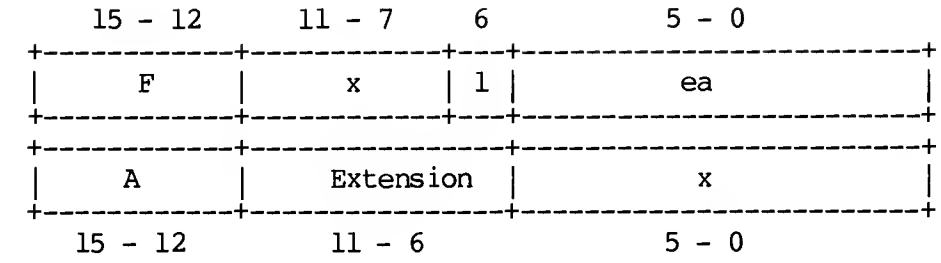
x — Unused, reserved

A.3.3.1.8 Conditional Branch



Extension	Predicate type 0 = Equal 1 = Not equal 2 = Greater or equal 3 = Less than 4 = Less or equal 5 = Greater than 6 = Ordered 7 = Unordered 8 - F = Unused, reserved
Offset	Branch offset: if field is zero, then a 16-bit offset follows in next word.
x	Unused, reserved

A.3.3.1.9 Set Byte on Condition



ea	Destination effective address
Extension	Predicate type 0 = Equal 1 = Not equal 2 = Greater or equal 3 = Less than 4 = Less or equal 5 = Greater than 6 = Ordered 7 = Unordered 8 - 3F = Unused, reserved
x	Unused, reserved

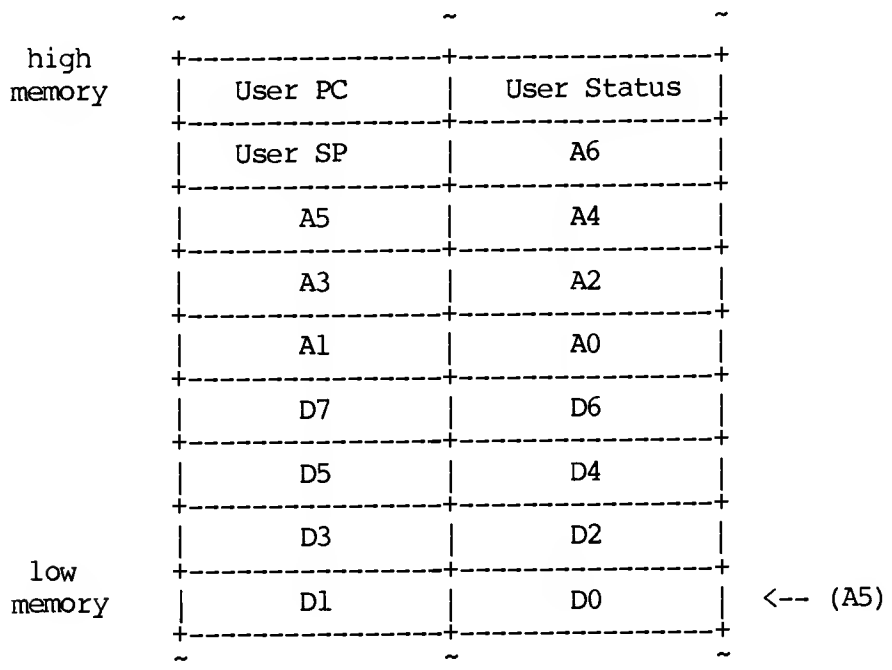
A.3.3.1.10 MC68000/MC68010 Effective Address Modification Words

Any words needed to further specify the MC68000/MC68010 effective address follow the FP instruction words.

A.4 CALLING SEQUENCE

The floating point software expects that it will be called through an F-line trap instruction. This instruction will initiate MC68000/MC68010 exception handling: the processor will switch to supervisor state, the current PC and status registers will be pushed onto the supervisor stack, and control will be transferred to the F-line trap handler routine. At this point, the trap handler should push the rest of the user's registers on the stack, including the user's stack pointer. The saved stack pointer value should point to the top of the stack that existed before the F-line instruction (or call to the F-line trap simulator) was executed. This requires adjusting the saved stack pointer if the F-line instruction is executed in the MC68000/MC68010 supervisor mode or if an F-line trap simulator is called in either MC68000/MC68010 user or supervisor mode since, in these cases, the stack on which the registers are saved is the same stack that the user's program had been using. This prohibits the use of autoincrement and autodecrement addressing modes with A7 in floating point instructions.

Register A5 should be set to point to the start of the entire user register state at the time of the F-line instruction. This stack-frame will appear as follows:



Additionally, the trap handler must set up the three address registers:

A1 - Pointer to the floating-point exception vector table
A3 - Pointer to the floating-point register block
A4 - Pointer to User memory fetch/store routine

At this point, a subroutine call to the floating point processor entry point may be executed.

On return from the floating point processor, the MC68000's or MC68010's registers must be restored from the stack. If the saved stack pointer had required correction, as described above, then the stack pointer should not be restored from the stack.

A.4.1 User Memory Fetch/Store Routine

All data references to the user's address space will be performed through a subroutine that must reside in the same address space as the floating point processor. The user memory routine will be called with the following registers defined:

```
A3 - Starting address in user memory
A2 - Starting address of floating point processor memory buffer
D6 - Program/data address space select
        0 = Program space
        1 = Data space
D7 - Low-order word contains:
        +n = Fetch n data bytes from user memory
        -n = Store n data bytes to user memory
```

All registers except A3, A2, and D7 should remain unchanged.

After the floating point processor has been called, it will call the user memory routine once to fetch each word of the F-line floating point instruction based on the user's PC. Once the effective address, if any, has been decoded from the floating point instruction, another call to the user memory routine will be made to transfer an operand in or result out to user memory.

Memory transfers of more than one byte should be performed on a word or long word basis, as appropriate, so that the user memory routine will detect address errors.

A.4.2 Floating Point Exception Handlers

When a floating point error occurs which sets a bit in the trap status byte and the corresponding bit in the trap enable byte is set, then the appropriate floating point exception handler will be executed. The vector to the exception handler is determined from a table of exception handler entry addresses or vectors. A pointer to the start of the vector table is passed to the floating point processor in A1. The vector table appears as:

```
low memory      Invalid operation vector  <-- (A1)
                Overflow vector
                Underflow vector
                Divide-by-zero vector
                Inexact result vector
                Integer overflow vector
                Reserved exponent vector
```

On entry to an exception handler, the stack pointer points to a return address which is followed on the stack by the stack frame, as described in paragraph A.4. The return address on the top of the stack is the return address to the F-line trap handler (or simulator) that called the floating point processor. Thus, to resume program execution, the exception handler only needs to execute a return from subroutine instruction. The return from subroutine returns control to the F-line trap handler (or simulator), which restores the MC68000/MC68010 registers and then returns control to the program.

The floating point processor does not force entry of the MC68000/MC68010 supervisor mode when it executes an exception handler. If the floating point processor had been called in the MC68000/MC68010 user mode from an F-line trap simulator, and the floating point processor detected the occurrence of an enabled exception, then the exception handler would be executed in the user mode.

PRE-CISION \ TYPE	NORMALIZED	ZERO	INFINITY	NaN	DENORMALIZED	UNNORMALIZED
SINGLE MEMORY	1 8 23 S | exp | signif 0<exp<$FF bias=127=$7F	1 8 23 S | 0 | 0	1 8 23 S | $FF | 0	1 8 1 22 X | $FF | t | addr X=undefined t=trapping NaN	1 8 23 S | 0 | ≠0	not possible
DOUBLE MEMORY	1 11 52 S | exp | signif 0<exp<$7FF bias=1023=$3FF	1 11 52 S | 0 | 0	1 11 52 S | $7FF | 0	1 11 1 51 X | $7FF | t | addr	1 11 52 S | 0 | ≠0	not possible
EXTENDED MEMORY	1 15 64 S | exp | 1.signif bias=0 $4000≤exp<$3FFF -16384≤exp<16383	1 15 64 S | $4000 | 0	1 15 64 X | $3FFF | 0	1 15 1 1 62 S | $3FFF | 0 | t | addr addr is right-justified	1 15 64 S | $4000 | 0.f (≠0)	1 15 64 S | exp | 0.f

FIGURE 1. Floating Point Number Format Summary

APPENDIX B

ASCII CHARACTER SET

CHARACTER	COMMENTS	HEX VALUE
NUL	Null or tape feed	00
SOH	Start of Heading	01
STX	Start of Text	02
ETX	End of Text	03
EOT	End of Transmission	04
ENQ	Enquire (who are you, WRU)	05
ACK	Acknowledge	06
BEL	Bell	07
BS	Backspace	08
HT	Horizontal Tab	09
LF	Line Feed	0A
VT	Vertical Tab	0B
FF	Form Feed	0C
CR	Carriage Return	0D
SO	Shift Out (to red ribbon)	0E
SI	Shift In (to black ribbon)	0F
DLE	Data Link Escape	10
DC1	Device Control 1	11
DC2	Device Control 2	12
DC3	Device Control 3	13
DC4	Device Control 4	14
NAK	Negative Acknowledge	15
SYN	Synchronous Idle	16
ETB	End of Transmission Block	17
CAN	Cancel	18
EM	End of Medium	19
SUB	Substitute	1A
ESC	Escape, prefix	1B
FS	File Separator	1C
GS	Group Separator	1D
RS	Record Separator	1E
US	Unit Separator	1F

ASCII CHARACTER SET (cont'd)

CHARACTER	COMMENTS	HEX VALUE
SP	Space or Blank	20
!	Exclamation point	21
"	Quotation mark (diaeresis)	22
#	Number sign	23
$	Dollar sign	24
%	Percent sign	25
&	Ampersand	26
'	Apostrophe, acute accent, closing single quote	27
(	Opening parenthesis	28
)	Closing parenthesis	29
*	Asterisk	2A
+	Plus sign	2B
,	Comma (cedilla)	2C
-	Hyphen (minus)	2D
.	Period (decimal point)	2E
/	Slant	2F
0	Digit 0	30
1	Digit 1	31
2	Digit 2	32
3	Digit 3	33
4	Digit 4	34
5	Digit 5	35
6	Digit 6	36
7	Digit 7	37
8	Digit 8	38
9	Digit 9	39
:	Colon	3A
;	Semicolon	3B
<	Less than	3C
=	Equals	3D
>	Greater than	3E
?	Question mark	3F

ASCII CHARACTER SET (cont'd)

CHARACTER	COMMENTS	HEX VALUE
@	Commercial at	40
A	Uppercase letter A	41
B	Uppercase letter B	42
C	Uppercase letter C	43
D	Uppercase letter D	44
E	Uppercase letter E	45
F	Uppercase letter F	46
G	Uppercase letter G	47
H	Uppercase letter H	48
I	Uppercase letter I	49
J	Uppercase letter J	4A
K	Uppercase letter K	4B
L	Uppercase letter L	4C
M	Uppercase letter M	4D
N	Uppercase letter N	4E
O	Uppercase letter O	4F
P	Uppercase letter P	50
Q	Uppercase letter Q	51
R	Uppercase letter R	52
S	Uppercase letter S	53
T	Uppercase letter T	54
U	Uppercase letter U	55
V	Uppercase letter V	56
W	Uppercase letter W	57
X	Uppercase letter X	58
Y	Uppercase letter Y	59
Z	Uppercase letter Z	5A
[	Opening bracket	5B
\	Reverse slant	5C
]	Closing bracket	5D
^	Circumflex	5E
_	Underline	5F

ASCII CHARACTER SET (cont'd)

CHARACTER	COMMENTS	HEX VALUE
`	Grave accent, Opening single quote	60
a	Lowercase letter a	61
b	Lowercase letter b	62
c	Lowercase letter c	63
d	Lowercase letter d	64
e	Lowercase letter e	65
f	Lowercase letter f	66
g	Lowercase letter g	67
h	Lowercase letter h	68
i	Lowercase letter i	69
j	Lowercase letter j	6A
k	Lowercase letter k	6B
l	Lowercase letter l	6C
m	Lowercase letter m	6D
n	Lowercase letter n	6E
o	Lowercase letter o	6F
p	Lowercase letter p	70
q	Lowercase letter q	71
r	Lowercase letter r	72
s	Lowercase letter s	73
t	Lowercase letter t	74
u	Lowercase letter u	75
v	Lowercase letter v	76
w	Lowercase letter w	77
x	Lowercase letter x	78
y	Lowercase letter y	79
z	Lowercase letter z	7A
{	Opening brace	7B
|	Vertical line	7C
}	Closing brace	7D
~	Tilde	7E
DEL	Delete	7F

APPENDIX C

68K-PASCAL LIMITATIONS

C.1 EXPRESSION COMPLEXITY

During Phase 1 of a Pascal compilation, expressions are translated to a reverse Polish form. The form uses a push-down stack for the operands, based on the precedence of the operators. If the precedence of the current operator is less than that of the next operator, pushing continues. The operators then operate on the top one or two operands on the stack, leaving the result on the top of the stack.

Phase 2 simulates the expression stack, using the processor's hardware stack and registers. It loads operands onto the stack -- actually into the processor's registers -- and then performs the appropriate operation. When the processor wants to load an operand but the registers are full, it pushes the "oldest" register onto the hardware stack in order to free a register.

To remember what is on the expression stack, Phase 2 maintains a 50-element array. Each element of the array describes one data item in registers or on the hardware stack. This limits the complexity of expressions that Phase 2 can handle to 50 levels of parentheses. When the array overflows, Phase 2 emits an error message of EXPR STACK OVERFLOW.

A scalar (integers, Booleans, characters, enumerated types, etc.) is put in a data register. A set of integers is put in two data registers; however, it takes up only one element of the 50-element stack array. Phase 2 allocates all eight data registers. A "standard" real value of any size is put in a floating point register. Phase 2 allocates a maximum of four floating point registers. Fast floating point values are put in registers D6 and D7. Pointers are put in address registers. Phase 2 allocates a maximum of four address registers. Strings, records, and arrays are always pushed directly onto the hardware stack. Each requires only one element of the expression stack array.

C.2 DATA STRUCTURES

A program's code size and data size are limited only by the amount of memory in the user's system (up to 16M bytes). The size of a component of a file type is limited to 32767 bytes due to the nature of the Pascal I/O utilities.

String constants are limited to a maximum of 132 characters. Strings are limited to 32766 bytes (32764 bytes of data and a 2-byte current length word). Sets are fixed at eight bytes.

The subranges of case statement index expressions and array index expressions may be any subrange which can be expressed using 4-byte integers.

C.3 LANGUAGE

ANSI Pascal states that a variant which becomes inactive will have all of its components totally undefined; therefore, it cannot be assumed that assignment to one variant field will define the fields of the other variants. Such an assumption could produce an unexpected result, illustrated in the case below. In this example, the intent is to perform arithmetic operations on the pointer by overlaying it in the record with an integer. However, due to register utilization, the compiler will generate code which uses the original value (10000) for the second pointer reference and not the intended value (10001).

```
TYPE pointrec = RECORD
                  CASE boolean OF
                     true : (i : integer);
                     false: (p : ^boolean)
                END;
VAR point : pointrec;
    b1,b2 : boolean;

BEGIN
    point i := 10000;
    b1 := point p^;
    point i := point i + 1;
    b2 := point p^;
    ...
END;
```

Global variables which are used in subprograms must all be defined in the same type, number, and order as in the main program, along with all global variables which precede them. The subprogram parameters must be the same as the program parameters in type, number, and order.

There is no runtime checking against MAXINT when an integer is read in.

Pascal does not check for overflow, even when runtime checking is enabled.

The Pascal statement: writeln; does not generate an empty line, as the Pascal standard requires, but generates a line containing a space character, because of a VERSAdos limitation.

If the log file from a batch operation was sent directly to a Centronics 703 printer, the intermediate line counter values in all three phases of the compiler are overprinted on top of the line counter headings. The final number of lines processed by each phase of the compiler is correctly printed. The command line option E should be used on the command line of each compiler phase when it is invoked from a batch file.

In response to an error which should produce a stack/heap error abort message (1010), one of the following abort messages may occur: bus error abort message (1008), address error abort message (1011), or illegal instruction abort message (8012).

APPENDIX D

ERROR MESSAGES

D.1 GENERAL

The Pascal Phase 1 listing and runtime messages are explained in paragraphs D.2 and D.3, respectively. Error code numbers not listed are not used.

D.2 PHASE 1 LISTING ERROR MESSAGES

When an error is discovered in a Pascal listing, the line following the error contains the message:

```
**ERROR--nnnnn**   ^xxx
```

where nnnnn is the line number where the last error occurred, ^ points immediately after the error, and xxx is one of the following numerical entries.

1: error in simple type
2: identifier expected
3: 'program' or 'subprogram' expected
4: ')' expected
5: ':' expected
6: illegal symbol
7: error in parameter list
8: 'of' expected
9: '(' expected
10: error in type
11: '[' expected
12: ']' expected
13: 'end' expected
14: ';' expected
15: integer expected
16: '=' expected
17: 'begin' expected
18: error in declaration part
19: error in field-list
20: ',' expected
21: '.' expected

50: error in constant

51: ':=' expected

52: 'then' expected

53: 'until' expected

54: 'do' expected

55: 'to' or 'downto' expected

58: error in factor

59: error in variable

101: identifier declared twice

102: low bound exceeds high bound

103: identifier is not of appropriate class

104: identifier not declared

105: sign not allowed

106: number expected

107: incompatible subrange types

109: type must not be real

110: tagfield type must be scalar or subrange

111: incompatible with tagfield type

112: index type must not be real

113: index type must be scalar or subrange

114: base type must not be real

115: base type must be scalar or subrange

116: error in type of standard procedure parameter

117: unsatisfied forward reference

119: forward declared; repetition of parameter list not allowed

120: function result type must be scalar, subrange or pointer

121: file value parameter not allowed

122: forward declared function; repetition of result type not allowed

123: missing result type in function declaration

124: fixed-point output format allowed for real only

125: error in type of standard function parameter

126: number of parameters does not agree with declaration

127: illegal parameter substitution

128: result type of parameter function does not agree with declaration

129: type conflict of operands

130: expression is not of set type
131: tests on equality allowed only
132: strict inclusion not allowed
133: file comparison not allowed
134: illegal type of operand(s)
135: type of operand must be Boolean
136: set element type must be scalar or subrange
137: set element types not compatible
138: type of variable is not array
139: index type is not compatible with declaration
140: type of variable is not record
141: type of variable must be file or pointer
142: illegal parameter substitution
143: illegal type of loop control variable
144: illegal type of expression
145: type conflict
146: assignment of files not allowed
147: label type incompatible with selecting expression
148: subrange bounds must be scalar
149: index type must not be integer
150: assignment to standard function is not allowed
151: assignment to formal function is not allowed
152: no such field in this record
153: type error in read
154: actual parameter must be a variable
155: control variable must neither be formal nor non-local
156: multi-defined case label
157: range of case indices too large
158: missing corresponding variant declaration
159: real or string tagfields not allowed
160: previous declaration was not forward
161: again forward declared
162: parameter size must be constant
165: multi-defined label
166: multi-declared label
167: undeclared label
168: undefined label

172: undeclared external file

175: missing file "input" in program heading

176: missing file "output" in program heading

177: assignment to function identifier not allowed here

201: error in real constant: digit expected

202: string constant must not exceed source line

203: integer constant exceeds range

207: exponent part of real constant contains too many digits

208: real constant exceeds range

250: too many nested scopes of identifiers

251: too many nested procedures and/or functions

255: too many errors on this source line

304: element expression out of range

305: element expression must not be real

306: expression not allowed in subrange set element

380: illegal character

381: constant string too long

390: set base type out of range

391: procedures and functions as parameters not implemented

392: file type not allowed here - as element of a structure

393: function or procedure not implemented

394: goto branching out of a procedure not allowed

395: file component type must be single identifier

396: writing of enumerated types to text files not allowed

397: selection on file buffer variable not allowed

400: illegal radix

401: number expected following radix and #

402: digit outside range allowed by radix

404: non-valid label

405: stack area (local or global) too large (over 16 Megabytes)

406: structure (record or array) too large (over 16 Megabytes)

407: invalid STRING size (maximum = 32764)

408: file component too large (maximum = 32767)

410: non-label found in exit statement

411: label not in current scope (exit statement illegal)

412: exit not inside loop construct

415: 'origin' expected

416: integer required after 'origin'

417: ']' expected - in 'origin' clause

420: no text allowed on same line after comment containing include file

Except for 503 and 511-515, the following "warnings" all appear as a result of the processor's W option (Table 2-1).

500: # not in standard Pascal

501: alpha label not in standard Pascal

502: unordered declarations not in standard Pascal

503: '{' encountered in '{' or '(*' encountered in '(*';
may be nested comments

504: 'origin' not in standard Pascal

505: 'exit' not in standard Pascal

506: 'otherwise' not in standard Pascal

507: 'subprogram' not in standard Pascal

508: structured function results not in standard Pascal

509: runtime file assignment not in standard Pascal

510: 'string' not in standard Pascal

511: syntax error in option comment

512: nesting of include files not allowed

513: only one include file allowed per comment

514: floating point type already specified

515: constant expression out of range

D.3 PASCAL RUNTIME ERROR MESSAGES

Pascal runtime error messages can occur whether a Pascal program is running on EXORmacs or VME/10 or on VERSAmodule 01:

a. On EXORmacs or VME/10 under VERSAdos, the following message is displayed:

```
aaaa: ABORTED BY bbbb=xyzz
```

where aaaa is the name of the task that is aborted, bbbb is the name of the task that caused aaaa to abort (typically, aaaa and bbbb will be the same task), and xyzz is the abort code (in hexadecimal). Abort codes are explained below. In addition to the abort code message, an explanation similar to the descriptions below is displayed.

b. On VERSAmodule 01, a program aborts by returning to VERSAbug. D0 contains $0E, and the lower two bytes of A0 contain the hexadecimal error code xyzz. (If VERSAbug is not resident, a STOP instruction is executed which lights the HALT and BRDFAIL LED's on the front edge of VERSAmodule 01.)

However, if the BREAK key was depressed, D0 then contains $FFFFFFFF and A0 is undefined. (If VERSAbug is not resident, the BREAK request is simply ignored.)

Normal (non-error) program terminations also return control to VERSAbug. D0 then contains $0F and A0 contains zeroes.

If xyzz has the form 1yzz or 2yzz, then it is a Pascal runtime error code as described below. Otherwise, it is an error code as described in the VERSAdos messages documentation (if a bus error or address error, it may be possible to successfully execute a program by specifying a larger stack/heap).

If xyzz is of the form 10zz, then it is one of the following error codes:

1001	Case index out of range
1002	Value out of range - found via range checking
1004	Integer division by zero
1008	Bus error - typically caused by invalid pointer value; can also be caused by stack/heap overflow
1010	Stack/heap overflow
1011	Address error - typically caused by invalid pointer; can also be caused by stack/heap overflow
1012	Memory allocation error during processing Z option
1013	Pascal data segment name must be 'SEG2'
1022	Read past end of file
1028	Illegal file name
1031	Integer expected - when reading from a text file
1032	Real expected - when reading from a text file
1033	Boolean expected - when reading from a text file

1040 Too many files in use or unrecognized device ID

1041 Option error in 'reset' or 'rewrite'

1042 Too many command line fields - maximum of 16 files + I + O may be specified

1043 File not open at input

1044 File not open at output

1051 Real number out of range - when reading from a text file

1052 Attempt to enable 6809 floating point trap

1053 Attempt to set 6809 floating point exception

1054 Attempt to set 6809 floating point precision mode

1062 Invalid base - when reading integer from a text file

1063 Invalid digit - when reading based integer from a text file

1070 Attempt to take NaN(0)

1099 Illegal TRAP 14 error code - internal Pascal error

NOTE: If the error is 1008, 1010, or 1011, it may be possible to execute the program successfully by running it with a larger stack/heap (specifying option Z).

If xyzz is of the form 2yzz, it indicates the occurrence of a standard floating point exception where y is the invalid operation code and zz indicates which exception(s) occurred.

The value of y (hexadecimal) is as follows:

0 No invalid operation error

1 Square root of a negative number, infinity in projective mode, or an unnormalized number

2 (+infinity) + (-infinity) in affine mode

3 Tried to convert a not-a-number to a binary integer

4 In division: 0/0, infinity/infinity, or unnormalized divisor and the dividend is not zero and is finite

5 One of the input arguments was a trapping not-a-number

6 Unordered condition tested by predicate other than equal or not-equal

7 Projective closure use of +/- infinity

8 0 x infinity

9 In 'rem': first argument is infinite or second argument is zero or unnormalized

A Input operand for binary-to-decimal or decimal-to-binary conversion out of range

B Tried to move a single precision unnormalized number to a double precision destination

C Tried to return an unnormalized number to single or double precision (invalid result)

The floating point exceptions that have occurred since the last time they were cleared are indicated in zz as a sum of the following (hexadecimal):

1	Invalid operation - see code in y above
2	Overflow
4	Underflow
8	Division by zero
10	Inexact result
20	Integer overflow - on conversion from floating point to integer
40	Reserved exponent value seen as input operand

If zz indicates that an invalid operation occurred (its lowest order bit is on) but the value of y is zero, then there was probably an error in one of the transcendental functions. This error could be any of the following:

a) Sine, cosine, or tangent of infinity or a not-a-number
b) Logarithm of a negative number, infinity, or a not-a-number
c) Arctangent of a not-a-number
d) Exponential of infinity or a not-a-number

SUGGESTION/PROBLEM REPORT

Motorola welcomes your comments on its products and publications. Please use this form.

To: Motorola Inc.
Microsystems
2900 S. Diablo Way
Tempe, Arizona 85282
Attention: Publications Manager
Maildrop DW164

Product: _____ Manual: _____

COMMENTS:

_____

_____

_____

_____

_____

_____

_____

_____

_____

Please Print

Name	Title
Company	Division
Street	Mail Drop / Phone Number
City	State / Zip

Field Service Support: (800) 528-1908
(602) 829-3100

MOTOROLA ***Semiconductor Products Inc.***

P.O. BOX 20912 • PHOENIX, ARIZONA 85036 • A SUBSIDIARY OF MOTOROLA INC.

16351-A PRINTED IN USA (9/83) MPS 2500